
ZX
Spectrum
ДИАЛЕКТЫ
БЕЙСИКА

ДИАЛЕКТЫ БЕЙСИКА

для



Под редакцией
Н. Родионова и А. Ларченко



«ПИТЕР»
Санкт-Петербург
1992

Диалекты Бейсика для ZX Spectrum

Под редакцией *Родионова Н. Ю., Ларченко А. А.*

Коллектив авторов: *Болдачев А. В., Думов И. Е., Елисеев В. А., Карпов П. А., Клюев Л. Е., Ларченко А. А., Родионов Ю. Н.*

Составитель *Болдачев А. В.*
Ответственный редактор *Усманов В. В.*
Редактор *Ромодин В. В.*
Художник *Непомнящий Д. Б.*
Корректор *Голубева З. Д.*

Оригинал-макет подготовлен *Болдачевым А. В.*

Книга посвящена языку программирования Бейсик для популярного компьютера **ZX Spectrum**. В ней подробно описаны: базовая версия Бейсика (**Spectrum-Бейсик**), наиболее распространенные расширения (**PRO-DOS**, **Laser Basic**, **MegaBasic**, **Beta Basic**), а также компиляторы (**ZX-Compiler**, **MCoder 2**, **Softek IS**, **Softek FP**, **Tobos FP**). Одна из глав содержит описание языка **Бейсик 128** для компьютера ZX Spectrum 128.

ISBN 5–7190–0001–1

Издательство «Питер».
194044, С.-Петербург, Выборгская наб., 17.
Подписано к печати 01.08.92. Формат 84 × 108 1/32.
Бумага типографская №1. Печать офсетная. Усл. п. л. 16,8.
Уч. изд. л. 17,2. Тираж 50 000. Заказ 192.
Отпечатано с готовых диапозитивов
в типографии им. Володарского.
191023, С.-Петербург, наб. р. Фонтанки, 57

ПРЕДИСЛОВИЕ

ZX Spectrum, или как его ласково называют на родине в Англии — **Speccy**, прочно вошел в жизнь многих наших соотечественников. Он оказался идеальным вариантом недорогого компьютера «для дома для семьи» (*home computer*). Маленький Speccy с огромным пакетом программного обеспечения, созданного за десять лет зарубежными программистами, стал своеобразной гуманитарной помощью Запада бывшему Союзу.

По традиции, сложившейся для машин класса home computer, один из популярнейших языков программирования **BASIC** (Бейсик) «вшит» в память ZX Spectrum. Это означает, что компьютер готов к работе на Бейсике сразу после включения.

Spectrum-Бейсик (так мы будем называть Бейсик, встроенный в ZX Spectrum) подробно, начиная с самых азов, описан в первой главе книги. Большая часть этой главы адресована тем, кто только учится программировать. Для более подготовленных читателей написан специальный раздел — справочник, где вся информация об операторах и функциях Spectrum-Бейсика приведена в сжатой и строгой форме.

Spectrum-Бейсик представляет собой набор лишь самых необходимых средств программирования. Поэтому сразу после появления Speccy стали создаваться программные пакеты, дополняющие Spectrum-Бейсик новыми операторами и функциями, — **расширения Spectrum-Бейсика**. Они подгружаются в память с магнитного носителя и открывают перед программистом множество дополнительных возможностей.

Каждое расширение Бейсика имеет свою область применения. Система **PRO-DOS** ориентирована на преобразование экранного изображения и вывод текста разнообразными шрифтами. Она используется, в основном, для создания демон-

страционных и рекламных программ. Но у PRO-DOS имеется существенный недостаток — она не позволяет перемещать картинки. На динамических эффектах специализируется пакет **Laser Basic**. С его помощью можно различными способами трансформировать графические объекты (спрайты) и передвигать их по экрану, создавая что-то наподобие мультфильмов.

При необходимости максимально расширить арсенал средств программирования можно пустить в ход «тяжелую артиллерию» — диалекты **MegaBasic** и **Beta Basic**. Они не только вводят много новых операторов и функций, но и делают более удобным и эффективным процесс написания и отладки программ. MegaBasic, в основном, ориентирован на усиление внешнего оформления программ (управление спрайтами, генерация звуковых эффектов и т. д.). Beta Basic незаменим для решения расчетных задач и реализации сложных алгоритмов обработки данных. MegaBasic и Beta Basic занимают в памяти компьютера довольно много места, но этот недостаток с лихвой окупается мощностью этих языков.

Одна из глав книги содержит описания **компиляторов Spectrum-Бейсика**, от самых простых (**ZX-Compiler**, **Mcoder 2**, **Softek IS**) до более серьезных (**Softek FP**, **Tobos FP**).

В последнее время все большую популярность среди синклеристов приобретает компьютер ZX Spectrum 128. Учитывая это, в книгу включена глава о языке программирования **Бейсик 128**, «вшитом» в память ZX Spectrum 128.

Глава, посвященная стандартному Spectrum-Бейсику, написана *А. Болдачевым*, консультировали его *Н. Родионов и А. Ларченко*. Компиляторы описали *В. Елисеев* и *Л. Клюев*, PRO-DOS — *Л. Клюев*, Laser Basic — *П. Карпов* (консультант — *А. Евдокимов*), MegaBasic — *Ю. Родионов*, Beta Basic — *И. Думов* (при участии *Л. Клюева*). Материал о Бейсике 128 подготовил *А. Ларченко*.

Условные обозначения, используемые в книге:

CS	–	клавиша **Caps Shift**;
SS	–	клавиша **Symbol Shift**;
CS/Z	–	клавиши нажимаются одновременно (в данном случае **Caps Shift** и **Z**);
Y+Z	–	клавиши нажимаются последовательно (в данном случае **Y** и **Z**);
[ ]	–	в квадратные скобки заключаются необязательные элементы формата операторов и функций;
|	–	может использоваться один из нескольких элементов формата, разделенных этим знаком.

SPECTRUM-БЕЙСИК

КОРОТКОЕ ВВЕДЕНИЕ

Это введение написано не для программистов, а для тех, кто только собирается ими стать. Конечно, новичку нужна особая книга, но, может быть, и эта начальная информация облегчит понимание не столь уж хитрого языка программирования — Бейсик.

Начнем с самого простого. Попробуем ответить на вопрос, что такое — компьютер? Что о нем должен знать программист?

Ничего особенного, кроме того, что в компьютере есть процессор, память и устройства ввода-вывода (клавиатура, монитор и другие). Достаточно знать, что процессор обрабатывает информацию, память хранит данные, а устройства ввода-вывода обеспечивают связь компьютера с внешним миром.

Основное же, что должно интересовать программиста — это список *команд процессора*. Ведь чем программист занимается? Лишь тем, что составляет из команд последовательности, называемые программами. И только.

Обобщим: программист должен хорошо представлять себе, как ввести в компьютер исходные данные, в каком виде можно получить результат, и великолепно знать, как при помощи команд преобразовывать одно в другое.

Список команд индивидуален для каждого типа процессора и закладывается раз и навсегда при его проектировании. Команды доводятся до процессора в виде чисел — *кодов*. Распознавая эти коды, он выполняет те или иные действия, например: пересылает данные из одного места памяти в другое, выполняет математические операции и другие элементарные преобразования. Последовательность, составленная из команд процессора, называется *программой в машинных кодах* или просто программой в кодах.

Оперировать с набором кодов неудобно, и естественное стремление программистов научить машину понимать человеческую

речь привело к появлению всевозможных *языков программирования* — систем, позволяющих записывать программы словами. Первым делом для составления программ стали использовать мнемоническую запись команд процессора — слова или наборы букв, однозначно соответствующие кодам. После того, как последовательность указаний процессору записана с помощью мнемоник, ее пропускают через специальную программу, которая переводит слова в коды. Такая система прямого перевода получила название *язык программирования низкого уровня — ассемблер*. «Слов» в ассемблере практически столько же, сколько команд процессора, выигрыш получается лишь в наглядности представления программы.

Следующим шагом в развитии языков программирования стала идея «записывать» часто повторяющиеся последовательности кодов одним, так называемым *ключевым словом*. Например, сказал компьютеру: «PRINT 2*2» (что в приблизительном переводе с английского означает: напечатай, сколько будет дважды два), а он и сосчитает, и выведет результат на экран, то есть выполнит достаточно длинную, но стандартную последовательность команд процессора. Ее надо только один раз составить, а потом использовать и для «дважды два — четыре» и для «трижды три — где-то семь».

Таким образом, достаточно написать множество стандартных последовательностей команд, выполняющих часто повторяющиеся операции, например: «вычислить косинус выражения», «опросить клавиатуру — не нажата ли клавиша?», «остановить выполнение программы», «если некое условие соблюдено, то выполнить заданные действия» и т. д. После этого каждую из последовательностей нужно назвать соответствующим человеческим словом: COS, INKEY, STOP, IF...THEN — и получится *язык программирования высокого уровня*.

Наиболее доступным из языков высокого уровня является Бейсик. Его исконное название — **BASIC**, а полностью — **Beginner All-purpose Symbolic Instruction Code**, что в переводе означает «инструктивный многоцелевой код для начинающих». Принято считать Бейсик языком учебным: он проще других в изучении и использовании. Но при этом Бейсик имеет все необходимые средства для решения большинства программистских задач. Впрочем, о Бейсике мы еще наговоримся, а сейчас еще несколько слов о языках программирования высокого уровня.

Итак, благодаря использованию в языках программирования «человеческих» слов программы стало писать легче и быстрее. И читать удобнее. Программисту. А компьютеру? Для компьютера надо переводить текст программы на компьютерный язык — язык команд процессора. Программа, понимающая слова языка программирования и заменяющая их соответствующими последовательностями команд процессора, называется *транслятор* (от англ. *translate* — переводить).

Наиболее простой способ трансляции — это немедленный «дословный» перевод, так называемая *интерпретация* (от англ. *interpret* — толковать). Программа, осуществляющая такой перевод, называется *интерпретатором*. Она опознает ключевые слова язы-

ка программирования (COS, STOP и прочие) и тут же «подсовывает» процессору соответствующую последовательность кодов.

Интерпретатор может работать в двух режимах: в непосредственном режиме и в режиме выполнения программы. В *непосредственном режиме* действие, заданное тем или иным ключевым словом, выполняется сразу же после введения слова с клавиатуры. Для работы в *режиме выполнения программы* предварительно словами языка программирования записывается последовательность требуемых от компьютера действий — *текст программы.* В тексте расписывается, что сделать сначала, что потом, что откуда взять и куда положить. Программа заносится в память компьютера и запускается на выполнение, то есть интерпретатору дается задание слово за словом считывать из памяти текст программы и переводить его на язык команд процессора. (Именно такой интерпретатор «зашит» в постоянной памяти ZX Spectrum.)

У интерпретаторов есть большое достоинство — простота отладки программ: выполнение программы в любой момент можно остановить и отредактировать ее текст. Но есть и существенные недостатки. Во-первых, программы работают только при наличии в памяти компьютера самого интерпретатора, который занимает довольно много места. И, во-вторых, программа, работающая с интерпретатором, делает все очень медленно. Время уходит и на распознавание слов языка, и на поиск соответствующих последовательностей команд процессора, и на прочее, прочее, прочее. Не говоря о том, что сами стандартные последовательности, вследствие их универсальности, получаются гораздо длиннее, чем программы в кодах, написанные специально для каждого конкретного случая. И, естественно, дольше выполняются.

От этих недостатков удалось избавиться с изобретением другого способа трансляции — *компиляции* (от англ. *compile* — собирать). *Компилятор* — специальная программа, исходным продуктом для которой, как и для интерпретатора, является текст на соответствующем языке программирования высокого уровня. Но в отличие от интерпретатора, компилятор не выполняет предписанные этим текстом действия немедленно, а создает, собирает из имеющихся в нем стандартных последовательностей другую программу — в кодах, не зависимую от каких либо трансляторов. Главная задача компилятора — сделать программу максимально быстродействующей. Откомпилированную программу, как и любую программу в кодах, трудно читать и изменять. Приходится редактировать исходный текст на языке высокого уровня и пропускать его вновь через компилятор, чтобы получить новую программу в кодах.

Созданы компиляторы и для Spectrum-Бейсика. О них позже. А сейчас — об интерпретаторе Бейсика. Основная часть первой главы рассчитана на начинающих. Для уже имеющих опыт в программировании предназначен специальный раздел — «Справочник по Spectrum-Бейсику» (далее просто «Справочник»).

ПЕРВОЕ ЗНАКОМСТВО

Бейсик — родной язык компьютера ZX Spectrum. На Бейсике он готов вести беседу сразу после включения питания и поймет то, что мы хотим ему сказать с полуслова. Ведь для ввода бейсиковской команды в Spectrum достаточно нажать лишь одну клавишу, ну две, в крайнем случае, три, не больше. И все благодаря тому, что программа, которая «сидит» в постоянной памяти компьютера, и клавиатура машины специально приспособлены для работы на Бейсике. Это видно при первом же знакомстве со Speccy: на его клавиши кроме букв, цифр и всяких значков нанесено еще по несколько английских слов. Это слова из лексикона Бейсика — имена его операторов и функций. Они называются *ключевыми словами*.

На большинстве компьютеров ключевые слова языков программирования, в том числе и Бейсика, набираются по буквам. Например, для набора слова PRINT приходится нажимать одну за другой пять клавиш: **P**, **R**, **I**, **N** и **T**. И при этом нельзя ошибиться, иначе компьютер не поймет, что, собственно, от него хотят. Speccy же, если с ним говорить на Бейсике, гораздо смышленее.

Включим компьютер и после того, как надоест любоваться фирменной заставкой

©1982 Sinclair Research Ltd

– нажмем клавишу, на которой нарисована буква **P**. В нижней части экрана, в так называемой *строке редактора*, мгновенно появится слово PRINT. Целиком и гарантированно без ошибок.

В результате нехитрого действия (нажатия клавиши) мы набрали, или, как принято говорить, *ввели* оператор PRINT. *Оператор* — это ключевое слово, предписывающее компьютеру выполнить некую законченную последовательность действий. Например, оператор PRINT отвечает за вывод на экран числовой и текстовой информации. Он так и переводится — «печатать». Что именно печатать, записывается следом за словом PRINT.

Там, на экране, за PRINT, давно уже мерцает квадратик с буквой L. Этот квадратик называется *курсором*. Его основное назначение — указывать место, куда будет введен символ или ключевое слово при нажатии клавиши. Нажмем, например, клавишу с изображением цифры **5**, и эта цифра выскочит на экран в том месте, где до этого был курсор, а сам курсор передвинется правее.

Теперь у нас получился полноценный оператор: ключевое слово с данными — PRINT 5. Можно приказать компьютеру выполнить его. Найдем на клавиатуре клавишу с надписью **Enter** (ввести, англ.) и нажмем ее. В верхнем левом углу экрана напечатается цифра 5 (что и ожидалось), сам оператор пропадет, а на его месте появится *сообщение*:

0 OK, 0:1

Мол, все о'кей, оператор* выполнен.

---

* Цифры **0:1** означают: первый в нулевой строке.

На этом примере мы не только ознакомились с тем, как одним нажатием клавиши набрать ключевое слово и выполнить получившийся оператор, но и с простейшим форматом оператора PRINT.

Формат — это узаконенный для языка программирования способ соединения ключевого слова, параметров и данных. *Параметры* задают режим работы операторов. Их количество и смысл строго определены для каждого из них. *Данными* будем называть «перерабатываемую» операторами информацию. В выполненном нами операторе данными является цифра 5, а параметров нет.

Оператор PRINT имеет более сложный формат, если с его помощью на экран нужно вывести не числовые данные, а *символьные*, то есть буквы или другие знаки (символьные данные часто еще называют *строковыми* или просто *строками*). В этом случае данные заключаются в кавычки. Вот так:

```
PRINT "p"
```

Набирается эта конструкция предельно просто. После включения компьютера (или выполнения какого-либо оператора) нажимаем клавишу **P** (последствия этого действия нам уже известны). Далее давим на клавишу с надписью **Symbol Shift** и, не отпуская ее, опять клавишу **P** — на экране появится кавычка ("). Потом еще раз — клавишу **P** и еще раз — **Symbol Shift** одновременно с **P** (кавычка). Теперь можно жать **Enter**, PRINT кавычки «обрежет» и напечатает на экране строчную букву **p**.

Если при наборе оператора допущена ошибка, нет необходимости сбрасывать компьютер и начинать ввод сначала. Пока не нажата клавиша **Enter**, набранную строку можно редактировать. Выведенные на экран ключевые слова и символы удаляются одновременным нажатием клавиш **Caps Shift** и **0**. Эта комбинация клавиш задает *команду редактора* **Delete**. После ее выполнения символ или ключевое слово, стоящее левее курсора, удаляется. Подвести курсор к любому месту набранной строки можно с помощью *клавиш перемещения курсора*: **Caps Shift** и **5** (влево) и **Caps Shift** и **8** (вправо). Если необходимо вставить в уже набранную строку еще что-нибудь, то к нужному месту подводится курсор и обычным образом набирается ключевое слово или символ. Текст, расположенный правее курсора, при этом сдвигается, освобождая место для вводимых символов. В этом коротеньком абзаце мы описали *редактор строк* Spectrum-Бейсика. Удобств в нем маловато, но набрать и поправить строку можно.

Пример с выводом на экран буквы **p** приведен здесь не только для того, чтобы продемонстрировать формат оператора PRINT при выводе символьных данных, но и для пояснения специфики работы клавиатуры ZX Spectrum. Для большей наглядности включим в строку, взятую в кавычки, еще прописную букву **P**, и значок ©. Они набираются нажатием все той же клавиши (рис. 1): прописная **P** — совместно с клавишей **Caps Shift**, а значок © — сложнее: сначала одновременно нажимаются уже изве-

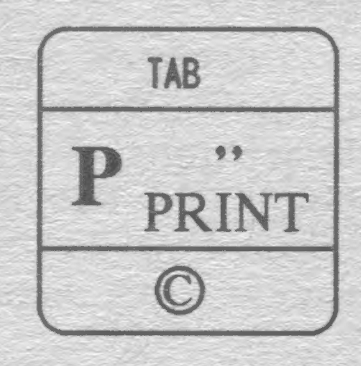


Рис. 1. Клавиша P.

стные нам клавиши **Caps Shift** и **Symbol Shift** и только потом **Symbol Shift и P**. Получим в строке редактора оператор

PRINT "pP©"

На этом примере мы показали, что, нажимая одну и ту же клавишу на клавиатуре ZX Spectrum в различных сочетаниях с двумя *управляющими клавишами* (**Caps Shift** и **Symbol Shift**), можно получить на экране разные символы. В примере мы вывели на экран и ключевое слово PRINT, и кавычку, и знак ©. На клавише **P** есть еще одно ключевое слово — TAB, которое также можно «вы-удить» с ее помощью. Пока без пояснений специфики использования TAB вставим его в оператор PRINT:

PRINT TAB 10; "pP©"

Для «извлечения» TAB нужно одновременно нажать клавиши **Caps Shift** и **Symbol Shift**, а потом клавишу **P**. Знак «точка с запятой» (;), расположенный на клавише **O**, необходим для нормальной работы оператора PRINT и набирается тем же способом, что и кавычка: совместным нажатием **Symbol Shift** и клавиши **O**. PRINT в сочетании с TAB напечатает на экране комбинацию из трех символов (pP©), но поместит ее не у левого края экрана, а ближе к его середине.

Такой «многоступенчатый» ввод — следствие отказа от побуквенного набора. Ведь ключевых слов Spectrum-Бейсика в несколько раз больше, чем клавиш у Speccy. А еще буквы, знаки. Вот и получается по несколько символов и ключевых слов на клавишу.

Что конкретно появится на экране после нажатия какой-либо клавиши, зависит не только от того, какие управляющие клавиши были нажаты до или одновременно с ней, но и от режима, в котором находится компьютер. Например, в одном режиме компьютер ждет ввода оператора или номера строки программы, в другом — набора цифр или букв. Мы уже убедились в этом: первый раз нажали клавишу **P** — получили ключевое слово PRINT, второй — буквочку p.

О своих ожиданиях, то есть о режиме, в котором он находится, компьютер дает знать по букве, помещенной в курсор (до ввода PRINT в курсор вписывалась буквочка **K**, сразу после ввода — **L**, а при наборе TAB и знака © — **E**). Индикация режима — это вторая функция курсора, в дополнение к обязанности указывать место, в которое будет поставлен символ либо ключевое слово при нажатии клавиши.

Учитывая все сказанное о клавиатуре, нам не избежать более подробного рассказа о том, как ею пользоваться. Вот такой уж Spectrum-Бейсик — без знания клавиатуры и программу не наберешь. Те, кто читает эту книгу из любопытства или только для общего знакомства, без особого ущерба для себя могут пропустить следующий раздел.

КЛАВИАТУРА

Режимов клавиатуры или так называемых *режимов курсора* пять. Обозначаются они буквами в соответствии с видом курсора: **[K]**, **[L]**, **[C]**, **[E]** и **[G]**.

Перед рассказом о клавиатуре договоримся, что управляющие клавиши **Caps Shift** и **Symbol Shift** далее будем обозначать, соответственно, **CS** и **SS**. Одновременное нажатие клавиш будем записывать через значок /, последовательное — через +. Например, порядок «извлечения» символа © запишется так: **CS/SS+SS/P**.

Режим ключевых слов (Keywords)

Курсор [K]

После включения или сброса компьютер устанавливает режим набора ключевых слов: выводит в нижний левый угол экрана курсор **[K]** и ждет ввода оператора. В этот режим он автоматически переходит и после ввода очередной строки программы или после набора двоеточия (:), а также после ввода ключевого слова THEN (см. стр. 33).

В режиме курсора **[K]** вводятся ключевые слова и знаки, размещенные в центральной части клавиши. К примеру, для клавиши **P** (см. рис. 1) — это PRINT и ". Ключевые слова, написанные чуть ниже центра клавиши (PRINT), вводятся простым нажатием клавиши. Ключевые слова или знаки, расположенные чуть выше центра (") — одновременным нажатием клавиши и **SS**. Цифровые клавиши в режиме **[K]** выводят цифры (простое нажатие), знаки (совместно с **SS**) и команды редактора* (совместно с **CS**).

Режим основных символов (Letter)

Курсор [L]

После появления на экране очередного ключевого слова компьютер заменяет курсор **[K]** на **[L]**, чем дает понять, что ждет ввода букв или цифр. Буквы Speccy печатает как большие (прописные), так и маленькие (строчные). Маленькие появляются при простом нажатии клавиши, а большие — при нажатии одновременно с управляющей клавишей **CS**.

В режиме курсора **[L]** при одновременном нажатии буквенной клавиши и **SS**, так же, как и в режиме **[K]**, на экране появляются символы и ключевые слова, расположенные чуть выше центра клавиши.

---

* Управление курсором, вызов на редактирование строки программы и т. д.

Режим прописных букв (Capital)

Курсор [C]

При необходимости напечатать длинную фразу большими буквами можно не удерживать клавишу **CS**, проще перейти в режим печати прописных букв. Для этого нужно нажать клавиши **CS/2** (**Caps Lock**). Курсор **[L]** заменится курсором **[C]**. Возвращается компьютер в режим курсора **[L]** при повторном нажатии **CS/2**.

Расширенный режим (Extend Mode)

Курсор [E]

В расширенном режиме на экран выводятся ключевые слова и знаки, написанные выше или ниже клавиш (либо — для многих самодельных вариантов ZX Spectrum — «на чердаке» и «в подвале» клавишной наклейки). Для перехода в этот режим нужно одновременно нажать управляющие клавиши **SS** и **CS**. Переход к расширенному режиму сопровождается появлением курсора **[E]**.

Верхние слова и символы вводятся простым нажатием клавиши в режиме **[E]** (например, TAB на клавише **P**), а нижние — с удержанием клавиши **SS** (символ ©). При извлечении нижних ключевых слов и символов, расположенных на нецифровых клавишах, вместо клавиши **SS** можно использовать и **CS**. После вывода слова или символа в расширенном режиме компьютер автоматически возвращается в режим курсора **[L]** (либо **[C]**, в зависимости от того, какой из них был включен до перехода в режим **[E]**).

Кроме перечисленных режимов работы клавиатуры (**[K]**, **[L]**, **[C]** и **[E]**), есть еще один — так называемый *режим псевдографики* (курсор **[G]** — **Graphics**). Об этом режиме мы расскажем в другом месте описания Spectrum-Бейсика (см. стр. 56).

После знакомства с клавиатурой можно не только читать эту книгу, но и «проигрывать» ее содержание на клавишах компьютера. Хотя делать это и необязательно. Постараемся так строить рассказ, чтобы читать книгу можно было не только дома, но и в метро или автобусе, куда не потащишь свой Speccy.

Несколько слов о последовательности изложения. Сначала в общем виде опишем основные операции, выполняемые Spectrum-Бейсиком (вывод на экран, математические вычисления, операции с символьными значениями, работа с графикой и др.). И лишь потом на простеньком примере приступим к собственно программированию. После этого расскажем о некоторых операторах и функциях необходимых, но относительно редко применяемых. Далее — описание системных переменных интерпретатора. В завершение несколько замечаний на темы «как сократить размер программы» и «как защитить ее от несанкционированного доступа». Решайте сами, что читать, что пропустить. Некоторые считают, что лучше сразу начать с программирования — попробуйте; если будет что непонятно, вернитесь к описанию основ.

ВЫВОД СИМВОЛОВ НА ЭКРАН

Невооруженным глазом видно, что изображения символов на экране Speccy строятся из отдельных точек. Эти точки в компьютерной терминологии называются *пикселями*. Символы вписываются в стандартные по размерам и положению площадки — *знакоместа*. Размер знакоместа — 8×8 пикселей. При соответствующем сочетании «зажженных» и «погашенных» пикселей в знакоместе вырисовывается тот или иной символ. В качестве примера приведем «строение» прописной буквы A стандартного набора символов ZX Spectrum (рис. 2).

Знакоместами как сеткой покрыт весь экран, образуя 24 строки и 32 столбца. Две последние строки Бейсик использует для набора и редактирования программ и вывода сообщений. Эти строки имеют собственное название — *служебный экран*. Остальные 22 строки называются *основным экраном* (рис. 2).

Каждое знакоместо имеет свою «прописку» — *координату*. Отсчет ведется от верхнего левого знакоместа, которому присвоены координаты (0, 0), — нулевая строка, нулевой столбец. С этого знакоместа начинает печатать первый выполненный после сброса компьютера оператор PRINT.

Если при печати текст заполняет все 32 знакоместа строки, то вывод символов автоматически переносится на начало следующей строки.

Каждый последующий PRINT начинает вывод символов с новой строки, если, конечно, не было никаких дополнительных указаний. Например, если предыдущий оператор PRINT завершился точкой с запятой (;), то следующий PRINT должен продолжить начатое дело и печатать со следующего знакоместа той же строки*:

```
PRINT "ZX";
```
ZX

```
PRINT " Spectrum"
```
ZX Spectrum



Рис. 2. Строение символьного экрана.

---

* В таком виде мы будем приводить примеры, выполняемые в непосредственном режиме: результат, выводимый на экран, будем выделять более жирным шрифтом. Указание на то, что для выполнения оператора нужно нажать клавишу **Enter**, а также сообщение о нормальном завершении работы (0OK) будем опускать.

Строка, выводимая вторым оператором PRINT, начинается знаком «пробел», название которого говорит само за себя — он очищает знакоместо. Для управления оператором PRINT кроме точки с запятой используются и другие знаки: запятая (,) и апостроф ('), но информацию о них «сошлем» в конец главы (см. стр. 57).

Основными средствами, указывающими оператору PRINT позицию экрана, с которой нужно начинать вывод символов, являются ключевые слова TAB и AT.

С TAB мы уже встречались и знаем, что после него ставится некое число — параметр. Параметр указывает знакоместо, в которое надо поместить первый выводимый на экран после TAB символ. Значение параметра соответствует порядковому номеру знакоместа в строке и может принимать значения от 0 до 31 (по количеству столбцов экрана).

Ключевое слово AT так же, как и TAB, используется совместно с оператором PRINT. Формат его подразумевает наличие после ключевого слова двух параметров, разделенных запятой. Значения этих параметров задают координаты знакоместа, с которого следует начинать вывод: первый — номер строки (0...21), второй — номер столбца (0...31).

Ключевые слова TAB и AT могут быть вставлены в любое место в строке после PRINT и должны отделяться от данных точкой с запятой:

```
PRINT TAB 9; "BASIC"; AT 0,0; "Spectrum-"
```

Spectrum-BASIC

МАТЕМАТИЧЕСКИЕ РАСЧЕТЫ

Алгебраические действия

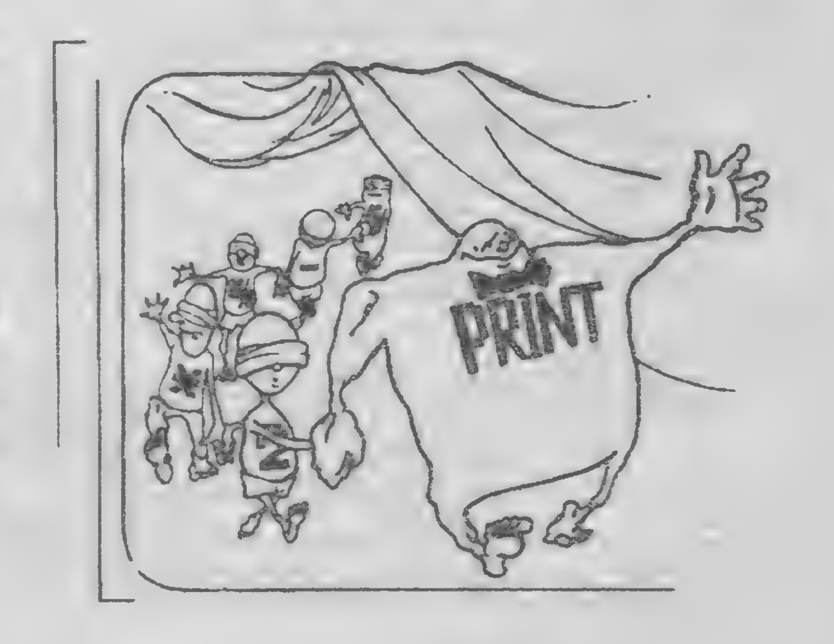
Первое представление, получаемое многими о компьютерах, сводится к пониманию того, что это нечто **считающее**. Такое представление недалеко от истины. Сам по себе компьютер только и умеет, что **считать**. Все остальное — происки программистов.

Для демонстрации математических способностей Speccy воспользуемся все тем же оператором PRINT. Выполним

```
PRINT 2+2
```

На экране появится цифра 4 — результат вычисления числового выражения, записанного после PRINT. Попробуем задать компьютеру задачку посложнее:

```
PRINT 2*(21.3/3–5.1)+2↑2
```



Он мгновенно выдаст результат — 8.

В этом примере были задействованы все *алгебраические операции*, понятные Spectrum-Бейсику:

- \+ сложение,
- − вычитание,
- * умножение,
- / деление,
- ↑ возведение в степень.

При записи чисел вместо запятой в десятичных дробях принято использовать точку. Скобки играют ту же роль, что и в обычных математических выражениях.

Последовательность вычислений также не отличается от принятой в математике: сначала подсчитываются значения выражений, взятых в скобки, внутри скобок в первую очередь выполняются операции возведения в степень, умножения, деления, потом сложения и вычитания.

Числовые функции

SQR, EXP, LN, SIN, COS, TAN, ASN, ACS, ATN

Функция в программировании — это специальная программа, преобразующая по заданному правилу некое исходное значение, называемое *аргументом*, в другое значение, называемое *результатом*. Принято говорить: «Функция возвращает результат».

Наиболее привычными и понятными функциями являются математические. Бейсик ZX Spectrum предоставляет в наше пользование набор самых необходимых из них:

SQR – квадратный корень,
EXP – экспонента,
LN – натуральный логарифм,
SIN – синус,
COS – косинус,
TAN – тангенс,
ASN – арксинус,
ACS – арккосинус,
ATN – арктангенс.

После каждого из перечисленных ключевых слов должен стоять аргумент функции — число либо выражение, которое может включать в себя в том числе и функции. Выражения необходимо заклю-



чать в скобки. Для примера, вычислим с помощью PRINT следующее выражение:

```
PRINT 1+COS PI—EXP (2—1.3)
—2.0137527
```

Здесь аргументом функции COS является константа PI. Вместо нее при вычислениях компьютер подставляет значение числа «пи», равного 3,1415927. Набирается PI обычным, принятым для ZX Spectrum способом — целиком ключевым словом (клавиши **CS/SS+M**).

Аргументы тригонометрических функций подставляются в радианах. Для перевода числа из градусов в радианы надо умножить его на PI/180, а для обратного преобразования умножить на 180/PI.

В алгебраических расчетах, кроме перечисленных стандартных функций, Spectrum-Бейсик позволяет использовать несколько специальных:

- ABS — вычисление абсолютного значения,
- INT — округление до ближайшего меньшего целого,
- SGN — определение знака аргумента.

Все функции Spectrum-Бейсика подробно описаны в «Справочнике».

ОПЕРАЦИИ С СИМВОЛАМИ

TO, LEN, STR$, VAL

Каждый язык программирования высокого уровня и, конечно же, Бейсик, кроме обработки числовых значений должен уметь преобразовывать и символьные данные — строки, составленные из букв, цифр и прочих знаков.

Простейшее действие, которое можно выполнить над символьными значениями, — это *сложить* их с помощью обычного знака +:

```
PRINT "BAS"+"IC"
BASIC
```

Результат такой же, как если бы мы попросили компьютер напечатать слово **BASIC** целиком (PRINT "BASIC").

В Бейсике ZX Spectrum можно выполнить операцию, обратную сложению: выделить из символьного значения некоторую его часть — *подстроку*. Это действие также называют *сечением*. Для его выполнения после строки символов в скобках через ключевое слово **TO** указывается, с какого и по какой символы требуется выделить подстроку:

```
PRINT "ABCDEF"(2 TO 5)
BCDE
```

Если подстрока выделяется, начиная с первого символа, то можно писать сокращенно: (TO 5). И, аналогично, если подстрока

выделяется до конца символьной строки, то разрешается записать: (2 TO). Допустимо задавать сечение и вообще без параметров:

```
PRINT "ABCDEF"(TO)
ABCDEF
```

Последнее сечение можно было записать и без TO:

```
ABCDEF()
```

Слово TO также можно опустить, если выделяется лишь один символ, то есть вместо (3 TO 3) использовать (3):

```
PRINT "ABCDEF"(3)
C
```

Если границы сечения выходят за пределы строки, компьютер выдаст *сообщение об ошибке*. Попробуйте ввести:

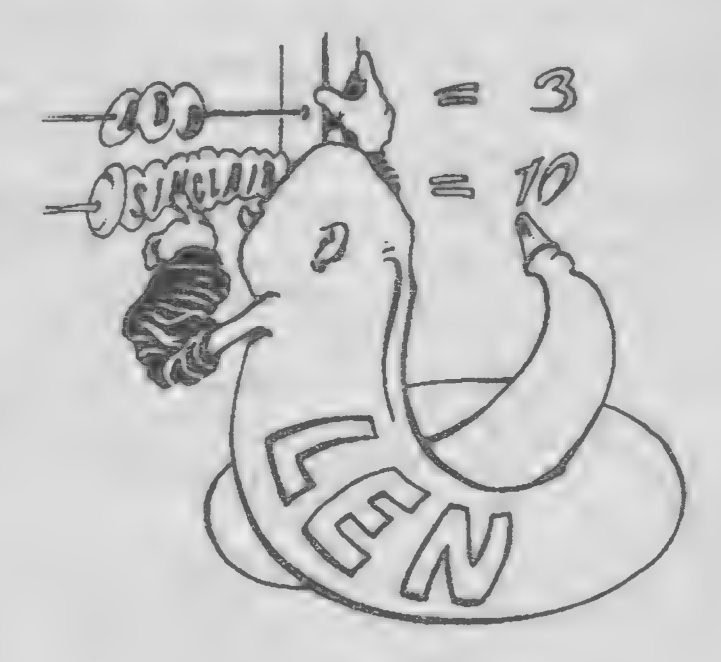
```
PRINT "ABCDEF"(2 TO 10)
```

– и получите внизу экрана сообщение Subscript wrong (неправильный индекс).

Вообще, если компьютеру что-нибудь не нравится в наших действиях, он будет ругаться различными английскими словами. Перевод сообщений об ошибках и описание ситуаций, в которых они возникают, можно посмотреть в «Справочнике».

С помощью функции **LEN** можно выяснить длину символьной строки, то есть определить, сколько позиций займет она на экране при печати*:

```
PRINT LEN "  SINCLAIR  "
10
```



Пробелы также считаются символами и учитываются при расчете длины.

Функция **STR$** преобразует числовые значения в символьные («прицепляет» к числу кавычки). Например, STR$ 3 возвращает символьное значение "3".

Можно рассчитать его длину:

```
PRINT LEN STR$ 3
1
```

При необходимости можно выполнить преобразование, обратное функции STR$, — превратить строковое значение в числовое. Нужно только воспользоваться функцией **VAL**:

```
PRINT VAL "5.25"–2.25
3
```

* Это не совсем точно: строка может содержать символы, которые не выводятся на экран.

VAL «оторвала» кавычки, превратив строковое значение в нормальное число, с которым можно производить математические операции.

Функция VAL может обрабатывать и числовые выражения, представленные как символьные значения. Например:

```
PRINT VAL "3*3+1"
10
```

Если VAL применить к символьному значению, не имеющему ничего общего с числами, то компьютер воспримет это как ошибку.



Символы и их коды

CODE, CHR$

Каждая буква, цифра, знак — вообще все, что мы называем символами, кроме визуального представления на экране (и клавиатуре), имеет еще и свой числовой эквивалент — *код*. Именно последовательность этих кодов сохраняется в памяти компьютера при записи символьного значения.

ZX Spectrum так организован, что ключевые слова Бейсика (PRINT, VAL и т. д.) также являются отдельными знаками, каждый со своим кодом*.

Всего в ZX Spectrum определено 256 символов с кодами от 0 до 255. Полный набор и тех и других можно посмотреть в таблице символов на стр. 100.

Выяснить, какой код соответствует символу или, наоборот, символ — коду, можно и не пользуясь таблицей, а с помощью функций **CODE** и **CHR$**:

```
PRINT CODE "A"
65
```



* Это существенно сокращает объем памяти, занимаемый бейсик-программой.

Так мы узнали, что код заглавной буквы **A** равен 65. Или, наоборот:

```
PRINT CHR$ 65
A
```

Что и требовалось проверить — коду 65 соответствует символ A.

Если символьное значение, подставляемое вслед за CODE, состоит больше чем из одного символа, функция возвратит код первого символа.

ГРАФИКА

PLOT, DRAW, CIRCLE, CLS

Оператор PRINT за одно элементарное действие — вывод символа — воздействует сразу на 64 точки экрана (8×8). Это, конечно же, хорошо с точки зрения ускорения процесса вывода символьной информации, но хотелось бы иметь возможность «обращаться» и к отдельным точкам (пикселям).

Spectrum-Бейсик говорит: «Пожалуйста», — и предоставляет в наше распоряжение оператор **PLOT**. С его помощью можно «включить» любой пиксель основного экрана*. Для оператора PLOT не существует разделения экрана на знакоместа. Для него экран — поле высотой 176 и шириной 256 пикселей. Каждый пиксель имеет свои координаты, не координаты знакомест в строках и столбцах, а *графические координаты*:

x — по горизонтали (0...255),
y — по вертикали (0...175).

Начало отсчета системы координат (пиксель с координатами x=0, y=0) находится в нижнем левом углу основного экрана. Обратите внимание на то, что отсчет вертикальной графической координаты идет снизу вверх, в отличие от нумерации символьных строк, которая производится от верха экрана (см. рис. 3).

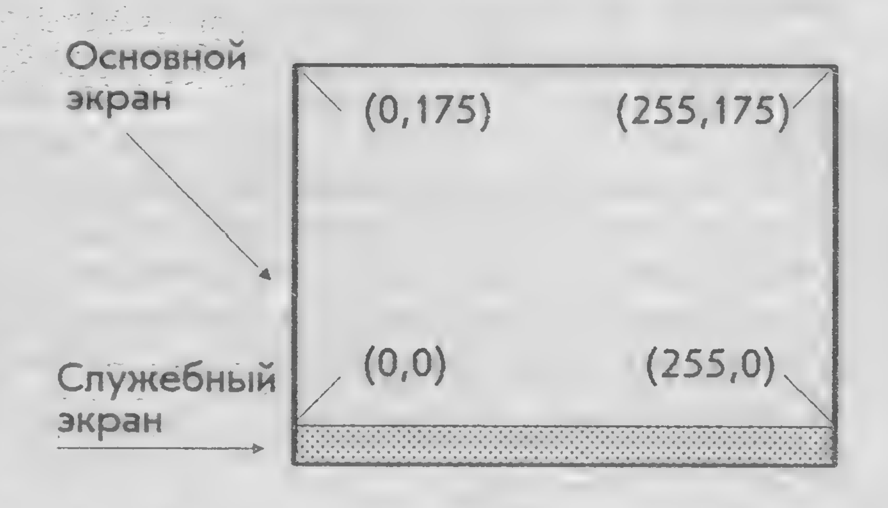


Рис. 3. Строение графического экрана.

После получения некоторого представления об устройстве графического экрана можно вернуться к оператору PLOT. Нетрудно предположить, что для работы ему нужны два параметра — координаты точки. Они записываются через запятую после ключевого слова: сначала x, потом y.

* Операторам, занимающимся графическими построениями, недоступны две строки служебного экрана.



К примеру, оператор PLOT 130, 80 высветит точку почти по центру экрана.

Ставя точку за точкой с помощью PLOT, в принципе, можно нарисовать чудесную картинку. Попробуйте. Однако вряд ли у кого-нибудь хватит терпения выставить более десятка точек. Кое в чем поможет сам Бейсик. Например, построение отрезков и дуг берет на себя оператор **DRAW** (от англ. рисовать, чертить).

DRAW при проведении прямой использует лишь два параметра: расстояние в пикселях по *x* и по *y* от некой начальной точки до конечной точки строящейся линии. Следовательно, параметры DRAW задают только направление и длину отрезка, но не устанавливают его начальную точку. Выполнив, например, оператор DRAW 127, 87 сразу после включения компьютера, увидим, что линия будет проведена к середине экрана от начала координат (0, 0). Следующий оператор, скажем, DRAW 0, −50 проведет отрезок длиной в 50 пикселей вертикально вниз от конца предыдущей линии. Вообще, началом линии для DRAW всегда является последняя выставленная на экране точка. Нагляднее и проще всего начальную точку можно задать с помощью оператора PLOT.

Если к оператору DRAW через запятую добавить еще один параметр, то на экране появится дуга. Этот дополнительный параметр задает величину угла (в радианах), который будет образован дугой. Знак параметра определяет направление построения дуги: положительный — против часовой стрелки, отрицательный — по часовой. Принцип получения начальной и конечных точек дуги остается тем же, что и при построении отрезка. Попробуем:

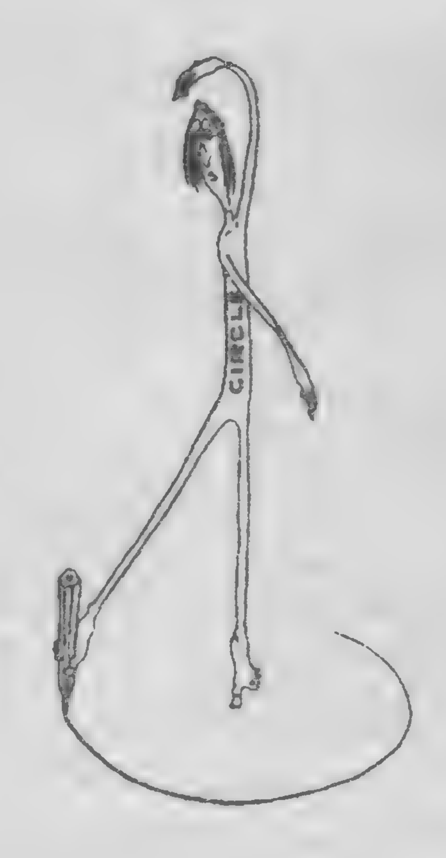
```
PLOT 150, 100
DRAW -100, 0, PI
```

Получим полуокружность с центром в точке (100, 100) и радиусом 50 пикселей. С константой PI мы уже сталкивались: она подставляет вместо себя число «пи».



Используя конструкцию с PLOT и DRAW, можно, конечно, построить и полную окружность. Но зачем, когда в Бейсике существует специально «обученный» для этого оператор **CIRCLE**? Его параметры очевидны: координаты центра и радиус окружности. Например, нарисуем окружность с центром в середине экрана и радиусом 50:

```
CIRCLE 127, 87, 50
```



Оператор CIRCLE строит окружности достаточно точно, но уж очень медленно.

Очистить экран от любого изображения (как от графики, так и от текста), то есть «выключить» все пиксели, можно оператором **CLS**, к описанию которого мы еще вернемся.

ЦВЕТ

Раскраска изображений

PAPER, INK, BORDER

Spectrum-Бейсик позволяет раскрашивать экран монитора с использованием восьми цветов: черного, синего, красного, фиолетового, зеленого, голубого, желтого и белого.

Изображение на экране строится из сочетания включенных и выключенных пикселей, причем как одни, так и другие Spectrum-Бейсик позволяет окрашивать в различные цвета. Цвет выключенных пикселей называется *цветом фона*, включенных — *цветом тона*. Следовательно, цвет фона — это цвет «пустого» (без изображения) экрана, цвет тона — цвет окраски символов или графических объектов, выводимых на экран.

Задаются цвета фона и тона, соответственно, операторами **PAPER** (бумага, англ.) и **INK** (чернила). После каждого из них ставится параметр — код требуемого цвета:

0 – черный,
1 – синий,
2 – красный,
3 – фиолетовый,
4 – зеленый,
5 – голубой,
6 – желтый,
7 – белый.



Например, после выполнения оператора

```
PAPER 6
```

знакоместа, в которые осуществляется вывод символов или графики, будут окрашиваться в желтый цвет.

Окрасить же весь основной экран можно, лишь очистив его с помощью оператора CLS. Он, выключая пиксели экрана, окрашивает их в текущий (заданный оператором PAPER) цвет фона. При работе в непосредственном режиме CLS можно заменить повторным нажатием **Enter**.

После окраски основного экрана становится четко виден *бордюр* — поле по краям экрана. Бордюр тоже можно окрасить в один из восьми цветов. Делается это с помощью оператора **BORDER**. Выполним:

```
BORDER 6
```

– и бордюр (так же, как и весь экран) станет желтым.

После задания цвета тона, например, оператором

```
INK 1
```

все выводимое на экран изображение — символы ли, графика ли — все будет окрашиваться в синий цвет. Попробуем:

```
PRINT "ZX Spectrum"
CIRCLE 127, 87, 50
```

На экране синим по желтому будет напечатано ZX Spectrum и начерчена окружность (по желтому в том случае, если до этого выполнялся предыдущий пример).

Цветовые эффекты

BRIGHT, INVERSE, FLASH, OVER

Изображение на экране Speccy может быть не только раскрашено, но и иметь две *градации яркости*: нормальную и повышенную (оператор **BRIGHT**). Кроме того, оно может выводиться с *инверсией*: цвет фона меняется на цвет тона, а цвет тона на цвет фона (**INVERSE**), или сделано *мерцающим*: с меняющимися через определенный промежуток времени цветами тона и фона (**FLASH**). Все перечисленные операторы используются с одним параметром, принимающим значение либо 1 — включение режима, либо 0 — выключение.

Это элементарно, но на пальцах не объяснишь. Лучше выполнить операторы BRIGHT 1, FLASH 1, INVERSE 1 и попробовать выводить на экран текст, задавая различные значения цветов тона и фона.

Обычно, если на экран помещается новое изображение, например, в знакоместо впечатывается символ, прежнее изображение стирается. Но в Spectrum-Бейсике имеется оператор **OVER**, после выполнения которого компьютер переходит в режим, позволяющий накладывать «новое» изображение на «старое». При этом точки, в которых изображения совпадают, принимают цвет фона. Включение и выключение режима наложения, как и режимов

инверсии и мерцания, задаются, соответственно, значениями 1 и 0, ставящимися после OVER.

Кроме упомянутых значений, указываемых после ключевых слов PAPER, INK (0...7) и BRIGHT, INVERSE, FLASH (0 и 1), может использоваться число 8, называемое *параметром прозрачности*. Например, после выполнения последовательности PAPER 8, INK 8, BRIGHT 8, INVERSE 8 и FLASH 8 оператор

```
PRINT AT 10,10;"*"
```

впечатает звездочку (*) в 10-й столбец 10-й строки в цветах, ранее установленных для этого знакоместа. Если, скажем, в этом знакоместе размещался мерцающий символ (FLASH 1) синего цвета (INK 1) на желтом фоне (PAPER 6) с повышенной яркостью (BRIGHT 1), то и звездочка будет выведена в этих же цветах.

После PAPER и INK может стоять также параметр 9. INK 9 предоставляет компьютеру самостоятельно выбрать цвет тона, который контрастен цвету фона: черным по светлому фону (зеленому, голубому, желтому и белому) или белым по темному (черному, красному, синему и фиолетовому). Таким же образом действует PAPER 9, устанавливая цвет фона контрастным цвету тона.

Постоянные и временные атрибуты экрана

Прежде всего поясним значение нового термина, появившегося в заголовке, — *атрибуты экрана*. Это общее название для цветов фона, тона, режимов яркости и мерцания, устанавливаемых для каждого знакоместа экрана. Задавая цвета и перечисленные режимы, мы говорим, что задаем атрибуты. Теперь поясним, почему они бывают постоянные и временные.

До сих пор мы использовали ключевые слова PAPER, INK, BRIGHT и FLASH как самостоятельные операторы. Их действие распространялось на весь экран и длилось до того момента, пока эти операторы не выполнялись с новыми параметрами. То есть мы работали с *постоянными атрибутами* экрана.

Однако часто бывает необходимо напечатать каким-либо цветом лишь одно слово или символ, не влияя на окраску последующих, то есть не меняя значения постоянных атрибутов. Сделать это можно, используя ключевые слова, управляющие цветом, в качестве параметров оператора PRINT.

Записываются они так же, как и AT и TAB — в произвольном месте данных, предназначенных для вывода на экран, отделяясь от них точками с запятой. К примеру, после выполнения строки

```
PRINT "*"; INK 4; "*"; PAPER 5; "*"
```

на экране появятся три звездочки. Цвета фона и тона первой из них будет соответствовать ранее определенным постоянным атрибутам (если они не были специально заданы, то звездочка будет напечатана черным по белому, то есть цветами, устанавливаемыми компьютером сразу после включения питания). Вторая звездочка окрасится в зеленый тон, но с тем же фоном. Третья пропечатается зеленым по голубому. Установленные в строке *временные атрибуты*, в данном случае цвета тона и фона, действуют

только до конца PRINT или до появления в строке новых параметров. Если после этого мы вновь зададим PRINT без параметров (PRINT "*"), то убедимся, что временные атрибуты никак не повлияли на постоянные.

Ключевые слова INVERSE и OVER тоже могут использоваться в операторе PRINT, переключая режим инверсии и совмещения лишь на время его работы.

Все графические построения (точки, линии, дуги, окружности) также можно производить не только в постоянных цветах (как до сих пор мы делали), но и во временных. Для этого операторы, задающие атрибуты экрана (INK, PAPER и прочие), ставятся сразу за ключевым словом графического оператора и отделяются от параметров точкой с запятой. Например:

```
PLOT 32, 80
DRAW 32, 0
DRAW INK 4; 32, 0
DRAW 32, 0
```

В результате выполнения этих операторов на экране высветятся три отрезка. Крайние будут окрашены в постоянный цвет тона (черный по умолчанию). Средний — в соответствии с временным параметром INK — в зеленый.

Но не все так радужно в королевстве цветной графики ZX Spectrum. Проведем по экрану две диагонали: красную и синюю. Чего проще:

```
PLOT 0, 0
DRAW INK 2; 255, 175
PLOT 255, 0
DRAW INK 1; –255; 175
```

Посмотрев же на экран, замечаем не очень приятный эффект: в районе точки пересечения первая проведенная диагональ (красная) перекрасилась в синий цвет.

Происходит это из-за того, что атрибуты экрана (цвета фона и тона, яркость и мерцание) в ZX Spectrum устанавливаются не для каждой точки, а целиком для знакоместа — одинаковые атрибуты для каждой из 64 его точек (8×8). И, естественно, если через знакоместо проведена линия с новым цветом тона, то все ранее установленные точки в этом знакоместе перекрасятся. Эта особенность компьютера накладывает ограничения на разнообразие окраски изображения: в пределах одного знакоместа невозможно использовать более двух цветов.

Операторы INVERSE и OVER также можно использовать и в качестве параметров графических операторов. Операторы с INVERSE 1 будут честно ставить точки или прочерчивать линии цветом фона, с OVER 1 на фоне будут рисовать тоном и, наоборот, на тоне фоном. Операторы, в которых одновременно размещены и INVERSE 1, и OVER 1, не будут менять изображение на экране.

ПРОГРАММИРОВАНИЕ

До сих пор, знакомясь с работой операторов Spectrum-Бейсика, мы вводили их в непосредственном режиме, то есть наше требование что-либо сделать компьютер выполнял сразу же после нажатия клавиши **Enter**. Для выполнения в этом режиме следующего оператора нужно опять набирать ключевое слово, параметры и опять нажимать **Enter**. Если бы компьютер мог работать только так, то он ничем принципиально не отличался бы от простейшего калькулятора, на котором каждое действие задается и выполняется отдельно.

Именно способность работать по программе, то есть выполнять один за другим операторы, расположенные в заданной последовательности, делает компьютер компьютером. Эта последовательность операторов и называется компьютерной программой.

Простейший способ заставить компьютер выполнить несколько операторов подряд без остановки — это после набора оператора поставить двоеточие, затем ввести следующий, а закончить последовательность нажатием клавиши **Enter**:

```
PRINT "A";: PRINT "B";: PRINT "C";[Enter]
ABC
```

Однако приведенная строка — еще не программа. Spectrum не может запомнить эту последовательность операторов и выполнить вновь. Да и не все операторы Бейсика можно записать вот так, в одну строчку. Не говоря уж о том, что читать такую последовательность не очень удобно.

Программа составляется из множества строк, в каждой из которых записан один или несколько операторов. Строкам присваиваются «адреса» — номера от 1 до 9999. При запуске программы строки выполняются одна за другой в порядке возрастания номеров.

Так, приведенная последовательность операторов PRINT в виде программы запишется следующим образом:

```
10 PRINT "A";
20 PRINT "B";
30 PRINT "C";
```

Набор программы

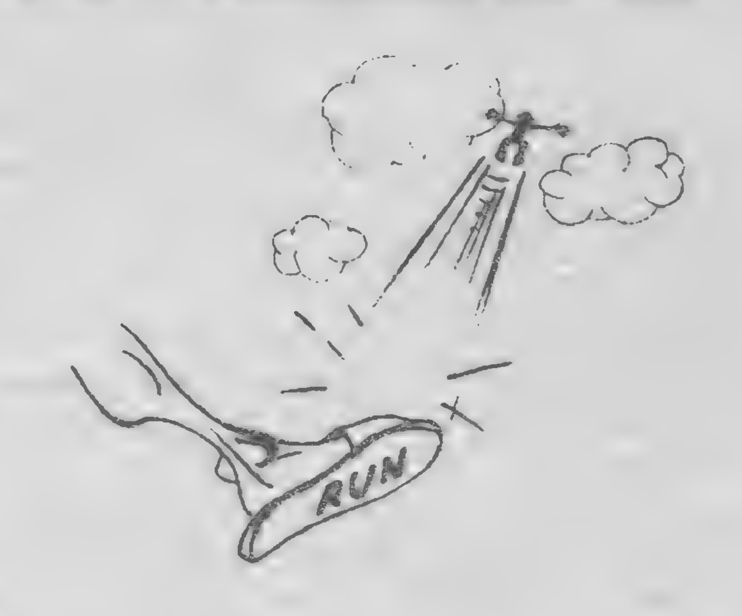
Строки программы набираются точно так же, как и операторы в непосредственном режиме, только перед первым оператором в строке обязательно ставится ее номер. После завершения набора нажимается клавиша **Enter**, но строка не выполняется, а переписывается в верхнюю часть основного экрана, освобождая место в строке редактора для ввода следующей. Именно номер в начале строки сообщает компьютеру, что набранный оператор или последовательность операторов являются частью программы, и их не следует выполнять сразу, а нужно поместить в память компьютера для хранения.

Запуск программы

RUN

Запускается программа оператором **RUN**, вводимым в непосредственном режиме. После выполнения нашей первой программы на экране увидим

ABC



и знакомое нам сообщение 0 OK, 30:1. Смысл его тот же, только цифры другие. Надеемся, понятные — работа закончена на первом операторе строки с номером 30.

RUN без параметров запускает программу с первой строки (со строки с наименьшим номером). Можно «стартовать» и с любой другой, указав ее номер. Например, оператор RUN 20 запустит программу с 20-й строки, и на экране появится лишь часть текста:

BC

Редактирование программы

LIST

При выполнении программы существенным является лишь порядок следования номеров строк. Поэтому для удобства последующего изменения программы (например, добавления новых строк) нумерацию производят через 5 или, чаще, через 10 номеров. Вставляются строки в программу просто: набирается строка с номером большим, чем номер строки, после которой нужно сделать вставку, но меньшим, чем номер последующей, и нажимается **Enter**. Введем строки

```
15  PRINT "*";
25  PRINT "*";
```

и наша первая программа приобретет следующий вид:

```
10  PRINT "A";
15  PRINT "*";
20  PRINT "B";
25>PRINT "*";
30  PRINT "C";
```

После ее выполнения (RUN) на экране появится

A*B*C

Допустим, нам потребовалось изменить содержание какой-нибудь строки программы (а этим только и приходится заниматься при ее отладке). Указать Бейсику строку, которая требует редакции, можно, перемещая с помощью курсорных клавиш **CS/6** и

CS/7 значок >, расположенный в тексте программы на основном экране между строкой и ее номером. Этот значок называется *указателем текущей строки* и, пока его «не трогают», указывает на последнюю введенную строку (в нашем случае на строку 25). Установим указатель на 30-ю строку и нажмем клавиши **CS/1** (**Edit**). Строка скопируется на служебный экран, туда, где она ранее набиралась и где ее теперь можно редактировать. Как это делается, мы уже описывали (см. стр. 9). Заменим, например, в этой строке букву **C** на **D**.

Когда все необходимые изменения в строке произведены, нажмем **Enter**. Отредактированная строка перепишется на прежнее место (если, конечно, не был изменен ее номер).

Пока строка находится на редакции (**Enter** не нажата), можно еще отказаться от внесенных в нее изменений. Нужно нажать **CS/1** (**Edit**), и на редактирование будет вызвана строка, в которой в данный момент расположен указатель текущей строки. Обратите внимание на то, что указатель текущей строки можно перемещать и во время редактирования строк.

Для удаления строки из программы совсем необязательно вызывать ее на редакцию и стирать весь ее текст. Достаточно ввести ее номер без каких-либо операторов. Введем, к примеру,

```
10
```

— и из программы исчезнет 10-я строка — и номера не останется. Пропадет с экрана и указатель текущей строки (возвращается он с помощью клавиш **CS/6** и **CS/7**).



И еще одна существенная, но простая операция, которую необходимо освоить для успешной отладки программы — вывод *листинга*, то есть распечатка на экране фрагментов программы. Это просто необходимо, если программа состоит из многих десятков, а то и сотен строк, которые, естественно, невозможно увидеть на экране сразу все. Начать вывод листинга с первой строки программы можно с помощью оператора **LIST**. После его выполнения компьютер представит на экране первую страницу листинга. Если на ней поместилась не вся программа, то он спросит: scroll? (прокрутить?). Для продолжения просмотра («прокрутки» экрана) нажимается любая клавиша, кроме **Space** и **CS/Space**. После нажатия на эти клавиши вывод листинга прекращается.

Можно попросить компьютер вывести листинг программы, начиная с любой строки, указав ее номер после ключевого слова, например:

```
LIST 20
```

Причем на упомянутую строку автоматически будет установлен указатель текущей строки. Таким образом, команда LIST позволяет быстро «перебрасывать» указатель в любое место программы.

Вполне может быть, что номер требуемой строки забылся. Не стоит расстраиваться. Если мы и ошибемся, LIST найдет строку с ближайшим большим номером и начнет вывод листинга с нее.

Переменные

LET

До сих пор мы работали с числовыми и символьными значениями, записанными в явном виде. Результаты действий не запоминали, а только выводили на экран монитора. В дальнейшей работе эти результаты использоваться не могли. Для вычисления выражения с новыми данными приходилось набирать и выполнять новую строку. То есть действовать так, как будто перед нами не компьютер, а калькулятор. Но даже в калькуляторах есть такая операция, как запоминание числа. Занесенное в память, оно может быть извлечено оттуда нажатием одной клавиши и использовано в дальнейших расчетах. И конечно же, написание даже самых простейших компьютерных программ немыслимо без сохранения исходных данных и промежуточных результатов. Причем запоминать надо не одно, не два числа, а часто сотни и тысячи. И не только числа, но и слова — *строки символов.*

Бейсик для хранения числовых и символьных значений в памяти компьютера отводит специальную область. Заносятся значения туда с помощью ключевого слова **LET**. А для того чтобы ничего не перепутать, каждому из «сосланных» на хранение значений присваивается собственное имя. Делается это так:




```
LET B=3
```

Этой строкой мы попросили компьютер записать в память число 3, а место, куда записано число, «обозвать» именем B. Теперь, что-

бы извлечь число, нужно просто «позвать» его по имени. Например, выполним оператор

```
PRINT B
```

– и на экране появится цифра 3. Имя можно подставить и в выражение. При вычислении значения выражения компьютер, наткнувшись на имя, заменит его числом из памяти:

```
PRINT (B+15)/B
6
```

Значение, занесенное в память, будет храниться там до тех пор, пока мы либо не сбросим (выключим) компьютер, либо не выполним операторы RUN или CLEAR (об этом операторе позже), либо не запишем под этим же именем другое значение. К примеру, выполним:

```
LET B=5
```

Именно потому, что записанные в память значения можно в любое время изменять, имена, под которыми они хранятся, называются *именами переменных.* Сейчас мы работали с переменной по имени B, или просто — с переменной B.

Обратите внимание на то, что переменная в языках программирования — это нечто другое, чем переменная величина в математических уравнениях. С точки зрения математики многие операции, производимые с программными переменными, бессмысленны, например, вот такая, часто используемая в программировании запись:

```
LET B=B+1
```

Приведенный оператор просит компьютер увеличить на единицу содержимое переменной B.

Переменные, предназначенные для хранения чисел, называются *числовыми.* Имя числовой переменной может состоять из любого количества цифр и латинских букв, но не должно начинаться с цифры. Его можно записывать и строчными, и прописными буквами. Пробелы в имени игнорируются. Приведем примеры имен числовых переменных:

```
A; A123; Rub; very long name; very long name
```

Для хранения символьных значений используются, соответственно, *символьные переменные* (их иногда также называют *строковыми*). Имя символьной переменной может состоять только из одной буквы и знака $ (доллар). Строчные и прописные буквы в имени не различаются. Присвоить символьной переменной значение можно с помощью того же ключевого слова LET:

```
10 LET A$="BASIC"
```

– а извлечь из памяти, подставив имя переменной, например, в PRINT:

```
20 PRINT A$
RUN
BASIC
```

Символьная переменная может быть приравнена *«пустой строке»*, то есть не содержать ни одного символа:

```
10 LET A$=""
20 PRINT "***"; A$; "###"
RUN
***###
```

То, что переменная A$ не содержит ни одного символа, хорошо видно по отсутствию интервала между символами *** и ###.

Все числовые и символьные функции будут нормально работать, если вместо конкретных значений им «подсовывать» имена переменных. Конечно, при условии, что до этого переменная была определена, то есть ей ранее уже присвоено какое-либо значение. Иначе компьютер выдаст сообщение об ошибке: Variable not found (переменная не найдена).

Теперь у нас вполне достаточно сведений, чтобы, наконец-то, перейти к написанию программ.

Составим элементарную программу, пересчитывающую некую сумму в рублях в доллары. По ходу ее написания мы будем знакомиться с новыми операторами и функциями Бейсика.

Рассчитаем сначала, сколько долларов составляет установленная нашим государством минимальная зарплата. В момент написания этой книги она равнялась 900 рублям в месяц. Введем переменные: Rub, в которую будем заносить сумму в рублях, Dol — для ее долларового эквивалента и Kurs — курс доллара, то есть цена одного доллара в рублях (переменную Kurs пока приравняем к 130, что соответствует нынешнему курсу доллара).

```
 5 REM Программа пересчета рублей в доллары*
10 LET Kurs=130
50 LET Rub=900: PRINT AT 0,1; "Текущий курс: "; Kurs; " рублей за
   $1"
70 LET Dol=INT(Rub/Kurs*100+0.5)/100: REM Расчет
75 PRINT AT 1, 1; Rub; " руб. — $"; Dol
```

Сам расчет предельно прост и умещается в коротенькой строчке под номером 70. Операция с умножением на 100, извлечением целой части функцией INT (см. стр. 16) и делением на 100 необходимы для округления долларовой суммы до центов. Остальные строки заняты вводом и выводом информации**.

Запустив программу, увидим на экране:

Текущий курс: 130 рублей за $1
900 руб. — $6.92

---

* В примерах программ, приведенных в этой книге, используется текст на русском языке. Делается это для наглядности, хотя не на всех компьютерах есть возможность его набрать. При вводе программ русские слова можно заменить английскими, а комментарии просто опустить.

** Если Вы решили отрабатывать все примеры на компьютере и хотите сохранить их на магнитной ленте, то загляните в раздел «Работа с магнитофоном».

Пояснения к тексту программы

REM

В 5-й строке «всплыл» новый оператор **REM** (от англ. *remark* — пометка). Он сообщает компьютеру, что в этой строке ему делать нечего, можно спокойно переходить к выполнению следующей. REM можно поставить и после двоеточия в конце любой строки, как это сделано в строке 70. Вслед за REM программист пишет для себя или для других программистов комментарии — пояснения, касающиеся работы программы.

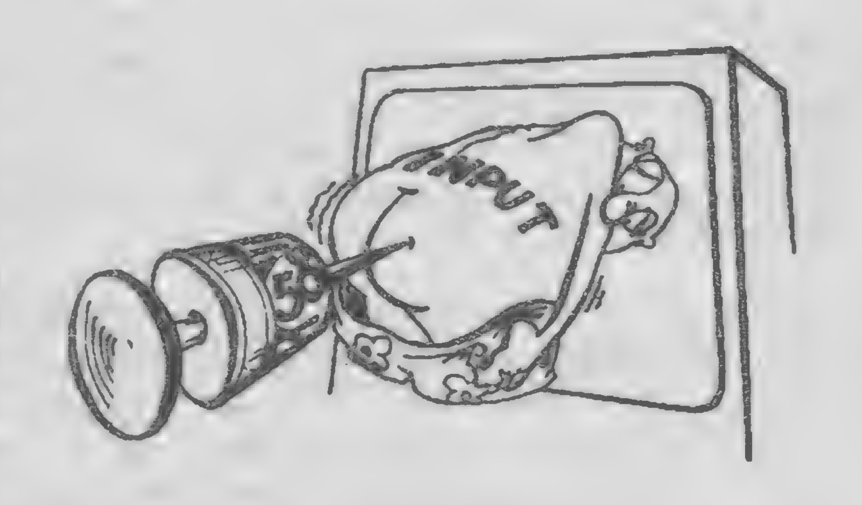
Ввод данных с клавиатуры во время выполнения программы

INPUT

Хотя написанная нами программа верно пересчитывает рубли в доллары, работать с ней неудобно. Для перевода другой суммы, да еще при постоянно меняющемся курсе, ее надо редактировать: изменять числа в строках 10 и 50 и снова запускать.

В ситуации, когда данные, необходимые для работы программы, нужно вводить по ходу ее выполнения, поможет оператор **INPUT**. В простейшем виде он записывается так:

```
10 INPUT Kurs
```



Натолкнувшись на него, компьютер останавливает выполнение программы, выставляет в последней строке экрана курсор и ждет, пока мы не введем с клавиатуры какое-нибудь число. После подтверждения ввода клавишей **Enter** оператор INPUT заносит введенное число в числовую переменную, имя которой указано вслед за ключевым словом (в нашем случае в переменную Kurs), и продолжает выполнение программы. INPUT не успокоится, пока не получит хоть какое-нибудь число. Если попытаться на запрос ничего не набирать, а сразу нажать **Enter**, то он укажет, что мы не правы — выведет знак вопроса (?). Можно, конечно, вместо числа предложить имя другой числовой переменной. Тогда переменной, стоящей вслед за INPUT будет присвоено значение набранной переменной, но только, если последняя ранее была определена. Иначе программа прервет работу с сообщением об ошибке: Variable not found.

Вслед за INPUT может стоять и несколько переменных. В этом случае оператор будет запрашивать числа столько раз, сколько переменных мы к нему прицепим:

```
10 INPUT Kurs; Rub
```

Возможности INPUT этим не исчерпываются. Кроме ввода данных он еще способен частично исполнять обязанности PRINT — выводить на экран текст. Используя это его свойство, мы можем заменить строки 10 и 50 нашей программы одной строкой:

```
10 INPUT "Текущий курс (руб. за 1$): "; Kurs; " Сумма (руб.): "; Rub
```

Формат записи данных в INPUT тот же, что и в PRINT: с теми же кавычками, с теми же знаками препинания. Так же с помощью PAPER, INK, BRIGHT, FLASH можно управлять атрибутами, а с помощью AT и TAB — позицией вывода.

Переходы

GO TO

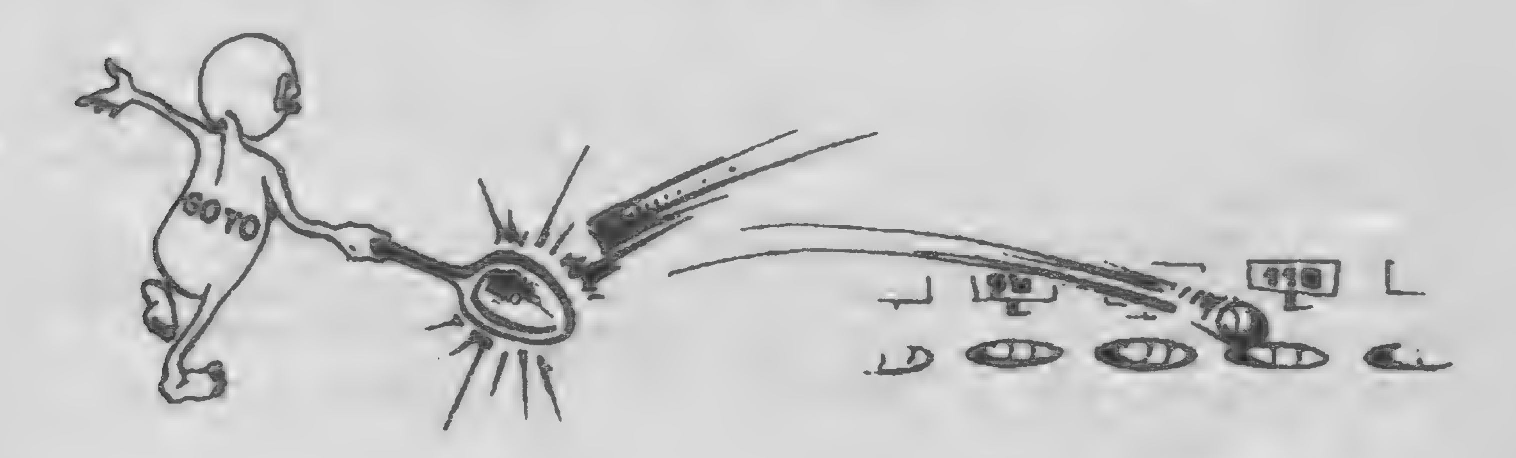
Есть еще одно неудобство — нашу программу после каждого расчета нужно запускать заново. Избавиться от этого предельно просто — стоит только ввести в действие оператор **GO TO**. Дословно он переводится «пошел на» строку с таким-то номером:

```
80 GO TO 10
```

Введя эту строку, мы, как говорится, «зациклим» программу: с последней строки отошлем компьютер опять к ее началу. Теперь без остановки программы можно пересчитать цены хоть всех товаров, выпускаемых в России. Но чтобы каждый раз не вводить курс доллара, все же разнесем ввод данных на две строки и переход по GO TO сделаем на строку 35:

```
 5 REM Программа пересчета рублей в доллары
10 INPUT "Текущий курс (руб. за 1$): "; Kurs
33 PRINT AT 0,1; "Текущий курс: "; Kurs; " рублей за 1$"
35 INPUT " Сумма (руб.): "; Rub
70 LET Dol=INT(Rub/Kurs*100+0.5)/100: REM Расчет
75 PRINT AT 1, 1; Rub; " руб. – ";"$"; Dol;"          "
80 GO TO 35
```

Пробелы (10 штук) в конце 75-й строки призваны затирать «хвост» выведенной в предыдущем расчете суммы.



Останов и продолжение выполнения программы

STOP, CONTINUE

Теперь возникла другая проблема: как остановить программу? Не выключать же компьютер. Можно, конечно, сделать сброс, но тогда сотрется программа.

Вообще, выполнение программы практически в любом месте можно прервать по команде **Break** (клавиши **CS/Space**). На это компьютер отреагирует одним из сообщений: BREAK into program (прерывание во время выполнения программы) либо BREAK – CONT repeats (прерывание - CONTINUE повторит) и после нажатия **Enter** перейдет в редактор Бейсика. Продолжить выполнение программы можно, введя оператор **CONTINUE**.

Вторую возможность остановить программу дает оператор INPUT. Если на его запрос вместо данных ввести оператор **STOP**, то с сообщением STOP in INPUT (стоп при вводе) мы «вывалимся» из программы. Как и после **Break**, продолжить выполнение программы можно по CONTINUE: вновь выполняется оператор INPUT.

Переходы по условию

IF...THEN

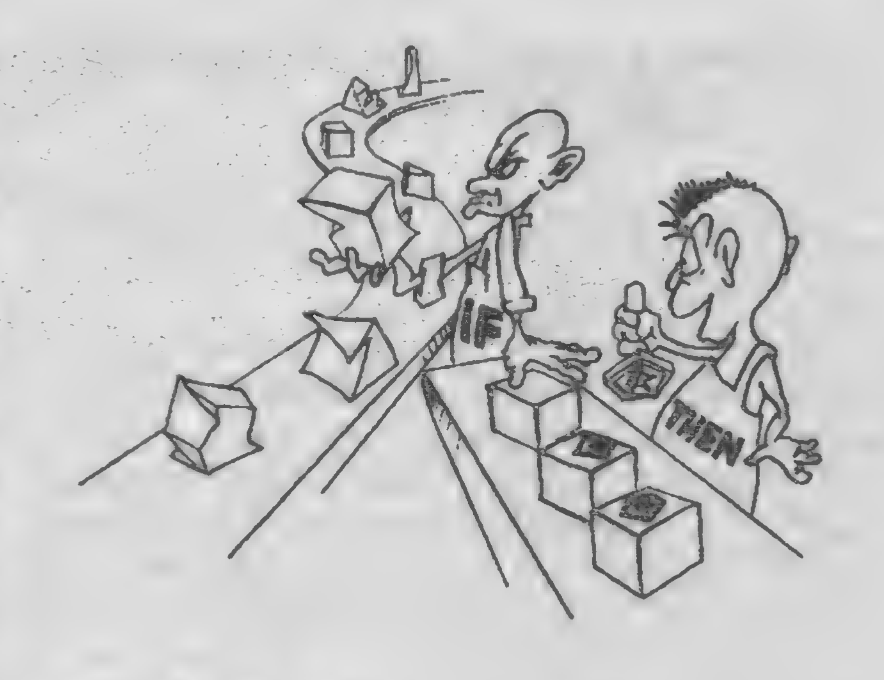
Требовать от пользователя, не посвященного в тонкости Бейсика (скажем, от продавца валютного магазина), чтобы он «вываливался» из программы по **Break** или STOP, по меньшей мере, несерьезно. Каждая программа должна корректно и просто завершаться. И информация о том, как это можно сделать, должна всегда присутствовать на экране. Например, в таком виде: «Для выхода из программы нажмите такую-то клавишу».

Поскольку наша программа постоянно ожидает получения нового числа, то проще всего организовать выход из нее при условии, что введено 0 руб. Следовательно, нам необходим оператор, способный проанализировать, равна ли переменная Rub нулю или нет. Если Rub равна нулю, то необходимо организовать выход из программы, а если Rub не равна нулю — пересчитать сумму в доллары и перейти к ожиданию ввода нового числа. Оператор, принимающий подобные решения, так и выглядит:

IF (если, англ.) **<условие>** **THEN** (тогда, англ.) **<действие>**

<Условием> может являться любое выражение. На принятие решения: выполнять или не выполнять действие, указанное за THEN, влияет результат вычисления этого выражения. Если значение выражения равно нулю, считается, что условие *ложно*, не равно нулю — *истинно*.

Чаще всего в качестве условия, стоящего за IF, используются выражения сравнения с использованием знаков «равно» (=), «больше» (>), «меньше» (<), «больше или равно» (>=), «неравно» (<>), «меньше или равно» (<=). (Обратите внимание на то, что знаки >=, <> и <= вводятся нажатием клавиш **Q**, **W** и **E** одновременно с **SS**).



<Действие> — это любой оператор Бейсика либо цепочка операторов, разделенных двоеточием. При истинности условия выполняться будет весь «хвост» строки следом за THEN.

В нашем примере условием является Rub=0, а действием — оператор STOP. Программа после включения новых строк приобретет такой вид:

```
10 INPUT "Текущий курс (руб. за 1$): "; Kurs
33 PRINT AT 0,1; "Текущий курс: "; Kurs; " рублей за 1$"
34 PRINT AT 15, 0; "Для окончания работы введите 0"
35 INPUT " Сумма (руб.): "; Rub
62 IF Rub=0 THEN STOP
70 LET Dol=INT(Rub/Kurs*100+0.5)/100: REM Расчет
75 PRINT AT 1, 1; Rub; " руб. — ";"$"; Dol:"          "
80 GO TO 35
```

Если условие, стоящее за IF, не выполняется (ложно), то компьютер, не обращая внимания на THEN, переходит к следующей строке программы.

Ввод символьных данных

INPUT, INPUT LINE

Конечная цель программиста — не написание программы, которая в принципе как-то работает, а создание такого продукта, который был бы защищен от ошибочных действий пользователя. Что, например, произойдет, если при работе с нашей программой ввести вместо числа букву (это можно сделать и чисто случайно, скажем, перепутав букву O с цифрой 0)? Программа «вывалится», но не нормальным образом, а с сообщением об ошибке: Variable not found. Должны мы предотвратить подобные ситуации? Конечно.

Компьютер не будет ругаться на нажатие любой клавиши, если попросить оператор INPUT ожидать ввода **символьного** значения. Сделать это просто — надо подставить в INPUT вместо числовой переменной символьную:

```
35 INPUT " Сумма (руб.): "; R$
```

На то, что ожидается ввод именно символьного значения, пользователю укажут кавычки, в которые INPUT заключит курсор и вводимый текст. От кавычек на экране можно избавиться, если перед символьными переменными, стоящими за INPUT, ставить ключевое слово LINE:

```
35 INPUT " Сумма (руб.): "; LINE R$
```

Хорошо, но в конечном итоге нам ведь необходимо получить число, а не символы.

Символьное значение, введенное с помощью INPUT LINE, преобразуем в числовое, используя функцию VAL:

```
60 LET Rub=VAL R$
```

К сожалению, этим проблема защиты от ошибочного ввода не решена. Ведь если теперь будет введено не числовое значение, то ошибку «выкинет» уже функция VAL. У нас остается один выход — проверить каждый введенный символ: является ли он цифрой или нет. Сделать это можно, анализируя коды введенных символов.

Для начала выделим как подстроку первый символ и определим его код:

```
40 LET Cod=CODE R$(1)
```

Далее проверим, попадает ли он в интервал от 48 до 57 (коды, соответствующие цифрам от 0 до 9, см. табл. 1 на стр. 100). Если символ не цифра, переходим на строку 100:

```
42 IF Cod<48 THEN GO TO 100
45 IF Cod>57 THEN GO TO 100
```

В 100-й строке мы в дальнейшем организуем обработку ошибки.

Логические операции

OR, NOT, AND

В этом месте немного отвлечемся от нашей программы и посмотрим, как можно две строки с номерами 42 и 45, проверяющие различные условия, заменить одной. Этот частный пример введет нас в пространство *логических операций.*

Логика оперирует с несколькими условиями и помогает делать вывод об истинности или ложности их сочетаний. Например, в нашей программе необходимо выяснить, выполняется ли хотя бы одно из двух условий: Cod<48 ИЛИ Cod>57. Нет ничего проще. Надо только русское ИЛИ заменить на английское, то есть на логический оператор **OR**:

```
Cod<48 OR Cod>57
```

Это объединенное условие истинно, если верно любое из сравнений. Учитывая, что истинное выражение принимает значение 1, а ложное — 0, можно записать:

x OR y = 1, если либо x=1, либо y=1, либо и x=1, и y=1
x OR y = 0, только если x=0 и y=0

Проверить, что введенный символ не цифра, можно и другим способом: выяснить, не попадает ли его код в промежуток от 48 до 57, то есть НЕ выполняются ли одновременно два условия: Cod>48 И Cod<57. Заменив НЕ на логический оператор **NOT**, а И — на **AND**, запишем:

```
NOT(Cod>48 AND Cod<57)
```

Выражение в скобках будет истинно только в том случае, если истинны оба условия. Так и запишем:

x AND y = 1, только если x=1 и y=1
x AND y = 0, если либо x=0, либо y=0, либо x=0 и y=0

Логический оператор NOT — это оператор отрицания. Из истинного утверждения он делает ложное и, наоборот, из ложного — истинное:

NOT x = 0, если x=1
NOT x = 1, если x=0

Вот и все логические операции, существующие в Spectrum-Бейсике. В качестве примера посмотрим, в каком случае может быть истинно утверждение мальчика Вовочки: «Я сегодня буду есть мороженое» (доллары пока в сторону). Итак, начнем:

(«Мама даст денег»
AND
((«будет работать магазин» AND «там будет мороженое»)
OR
(«будет работать киоск» AND «там будет мороженое»))
AND NOT
«пойдет дождь»)
OR
(«мороженое принесет папа»
AND NOT
«мама оставит мороженое в холодильнике до завтра»)

Вот такая ЛОГИКА.

Теперь вместо двух строк нашей «валютной» программы, проверяющих, что введенный символ — цифра, можно записать одну:

```
45 IF CODE R$(1)<48 OR CODE R$(1)>57 THEN GO TO 100
```

В этой строке мы избавились от лишней переменной Cod.

Такую проверку надо организовать для каждого символа переменной R$, то есть продолжить:

```
46 IF CODE R$(2)<48 OR CODE R$(2)>57 THEN GO TO 100
```

Однако если кто-нибудь захочет перевести в доллары, скажем, 3 рубля (ха-ха!), и, следовательно, переменная R$ будет состоять всего из одного символа, то программа в строке 46 «вылетит» по ошибке (параметр в сечении превысит пределы допустимого). В подобных случаях необходимо сначала определить длину символьной переменной и выполнить проверку столько раз, сколько символов в переменной R$, то есть организовать *цикл*.

Программные циклы

FOR, TO, STEP, NEXT

Введем переменную i, называемую *счетчиком цикла*, присвоим ей значение 1 и подставим в строку 45 для выделения первого

символа переменной — R$(i). После проверки кода символа увеличим значение i на 1 и сравним его с длиной переменной R$ (предполагается, что R$ не равна «пустой строке»). Если i еще не превысила LEN R$, то перейдем снова на строку 45, выделим из R$ следующий символ (ведь i увеличилась на 1) и проверим его код:

```
40 LET i=1
45 IF CODE R$(i)<48 OR CODE R$(i)>57 THEN GO TO 100
48 LET i=i+1
50 IF i<=LEN R$ THEN GO TO 45
```

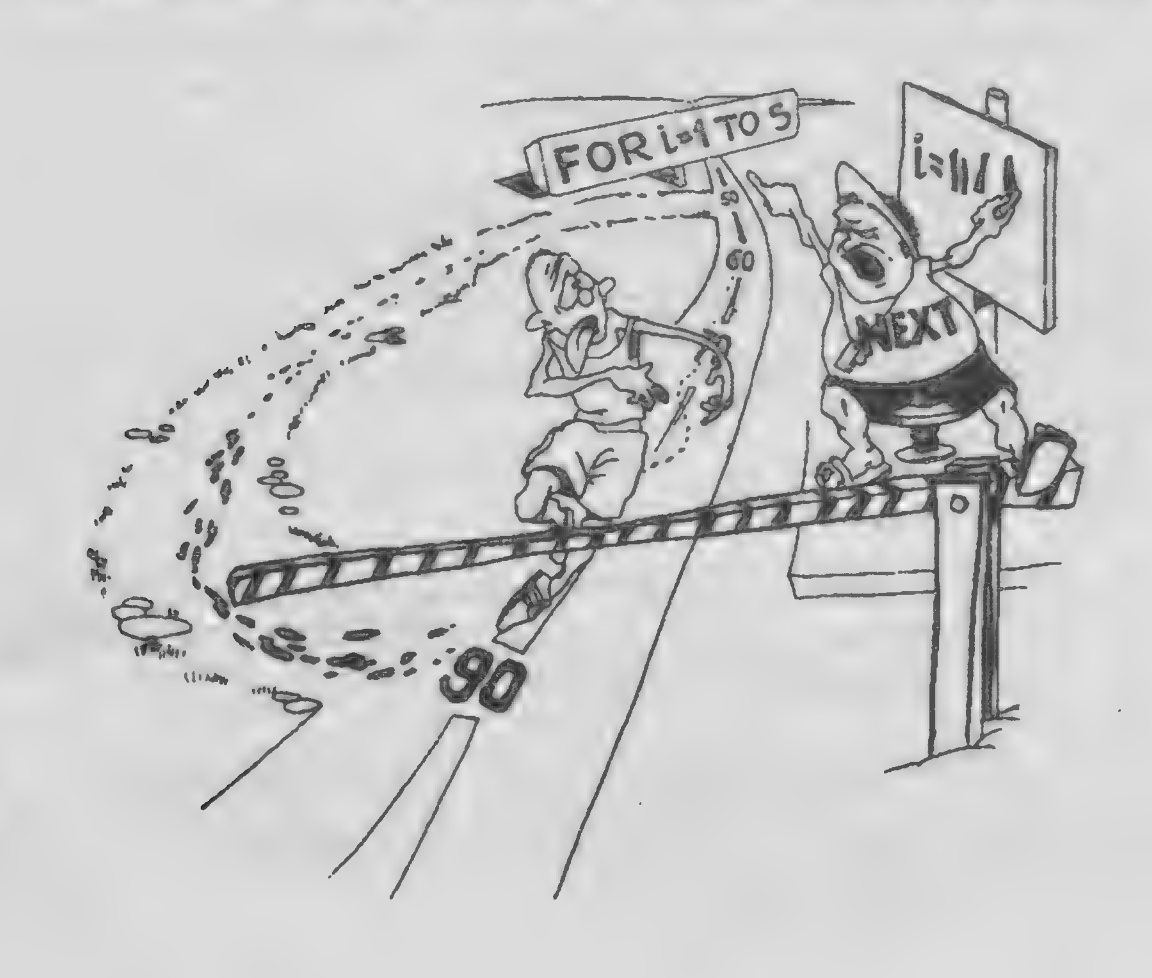
Выход из цикла произойдет, когда i превысит количество символов во введенной строке (i>LEN R$).

Конструкция, подобная построенному нами циклу, настолько часто применяется в программировании, что для ее организации в Бейсик введены специальные операторы **FOR** и **NEXT**. С их помощью наш цикл запишется следующим образом:

```
40 FOR i=1 TO LEN R$
45 IF CODE R$(i)<48 OR CODE R$(i)>57 THEN GO TO 100
50 NEXT i
```

Следом за ключевым словом FOR следует операция присвоения переменной i *начального значения* — эквивалент LET i=1. После знака равенства может стоять числовое выражение. Переменная, использующаяся в цикле FOR...NEXT в качестве счетчика, носит специальное название — *управляющая переменная цикла.* Это обычная числовая переменная, но с единственным ограничением: ее имя должно состоять из одной буквы.

За ключевым словом TO указывается *конечное значение* переменной цикла (число либо числовое выражение). Сравнение переменной цикла и предельного значения осуществляет оператор NEXT, который ставится в конце цикла с указанной вслед за ним управляющей переменной. Строка с NEXT i аналогична прежним строкам



48 и 50. NEXT увеличивает переменную цикла и сравнивает ее с предельным значением. Если значение переменной цикла превысило конечное значение, то выполнение программы продолжается с оператора, следующего за NEXT. Иначе делается еще один «виток»: повторяются все операторы после FOR.

В нашем примере переменная цикла с каждым проходом увеличивается на 1, но значение *шага цикла* можно варьировать, введя ключевое слово **STEP** и указав за ним значение, на которое следует увеличивать переменную цикла при выполнении оператора NEXT.

Строку 40 можно записать и так:

```
40 FOR i=1 TO LEN R$ STEP 1
```

Правильность набора нужно проверить не только при вводе суммы, но и при вводе значения курса доллара. Придется организовать еще один цикл, подобный уже описанному. Включим в программу и строку 100, выводящую сообщение о допущенной ошибке, и строку 105, «отсылающую» к строкам с оператором INPUT, в которых была допущена ошибка ввода. Номер строки, на которую следует перейти, заносится в переменную N.

```
  5 REM Программа пересчета рублей в доллары
 10 INPUT "Текущий курс (руб. за 1$): "; LINE K$
 13 IF LEN K$=0 THEN GO TO 10
 15 FOR i=1 TO LEN K$
 20 IF CODE K$(i)<48 OR CODE K$(i)>57 THEN LET N=10: GO TO 100
 25 NEXT i
 30 LET Kurs=VAL K$
 33 PRINT AT 0,1; "Текущий курс: "; Kurs; " рублей за 1$"
 34 PRINT AT 15, 0; "Для окончания работы введите 0"
 35 INPUT "Сумма (руб.): "; LINE R$
 37 IF LEN R$=0 THEN GO TO 35
 40 FOR i=1 TO LEN R$
 45 IF CODE R$(i)<48 OR CODE R$(i)>57 THEN LET N=10: GO TO 100
 50 NEXT i
 60 LET Rub=VAL R$
 65 IF Rub=0 THEN STOP
 70 LET Dol=INT(Rub/Kurs*100+0.5)/100: REM Расчет
 75 PRINT AT 1, 1; Rub; " руб. — ";"$"; Dol:"          "
 80 GO TO 35
100 PRINT AT 10, 0;"Извините. Вы ошиблись в наборе."
101 PAUSE 50
103 PRINT AT 10, 0;"                                "
105 GO TO N
```

Вот теперь компьютер прервет выполнение программы только тогда, когда мы введем сумму 0 руб. (либо оператор STOP). При ошибочном вводе компьютер извинится и предложит повторить набор.

Строки 13 и 37 предохраняют от ввода «пустой строки», то есть от нажатия **Enter** без ввода символов. Если их пропустить, то в строках 30 и 60 программа «вылетит» по ошибке, не сумев выполнить VAL от «пустой строки».

Пауза

PAUSE

В строке 101 программы использован оператор **PAUSE**. Он приостанавливает выполнение программы. Длительность паузы задается параметром оператора в 1/50 долях секунды. Оператор PAUSE 0 задает бесконечную паузу. Прервать паузу можно и раньше установленного срока, нажав любую клавишу.

В нашей программе пауза нужна для того, чтобы пользователь успел прочитать сообщение об ошибке. После чего сообщение затирается пробелами (строка 103).

Опрос клавиатуры

INKEY$

Есть еще одна возможность сделать нашу программу более приятной в общении. При выходе из программы принято запрашивать подтверждение желания пользователя закончить работу. Ведь возможна ситуация, когда на запрос случайно будет введен ноль. Программа прервется, хотя мы этого и не хотели.

Обычно, при необходимости запросить подтверждение какого-либо действия, программу «зацикливают» на опросе клавиатуры. Решение «что делать» принимается в зависимости от того, какая нажимается клавиша. Для опроса клавиатуры применяется функция **INKEY$**. Она не требует аргумента и возвращает символ, соответствующий клавише, нажатой в данный момент. Если никакая клавиша не была нажата, функция возвращает значение «пустая строка».



Без предисловий напишем необходимый нам фрагмент программы:

```
120 PRINT AT 10, 5; "Вы действительно хотите закончить работу? Y/N
    (Да/Нет)"
140 IF INKEY$="" THEN GO TO 140
145 IF INKEY$="Y" OR INKEY$="y" THEN STOP
150 CLS: GO TO 5
```

Пока никакая клавиша не нажата, программа «зациклена» на строке 140. Если нажата клавиша **Y** (с **Caps Shift** или без нее),

программа останавливается. По любой другой клавише осуществляется переход к началу программы.

Звук

BEEP

Никогда нелишне украсить программу одной-двумя музыкальными фразами.

Spectrum-Бейсик не позволяет создать многоголосое музыкальное сопровождение, но «насвистеть» мелодию сумеет. Для этого имеется специальный оператор **BEEP**. Наберем ключевое слово BEEP, вслед за ним через запятую два числа, скажем, 1 и 0:

```
BEEP 1, 0
```

В течение одной секунды мы будем слышать звук с частотой, соответствующей ноте ДО первой октавы. Меняя первое число, можно регулировать продолжительность звучания ноты (значение задается в секундах). Второе число определяет высоту звука: увеличение числа на 1 соответствует переходу на пол тона выше, уменьшение на 1 — на пол тона ниже. Для поднятия ноты на октаву ко второму параметру BEEP следует прибавить 12.



Указанный принцип отсчета параметра BEEP не точен: высоты нот при его использовании не будут совпадать с принятыми в музыке. Чтобы не травмировать слух окружающих, необходимо использовать более точные параметры. На рис. 4 приведено точное соответствие значений второго параметра оператора BEEP нотам первой октавы.

Для проигрывания мелодии, естественно, нужно написать программку, состоящую из последовательности BEEP. Вот как, к приме-

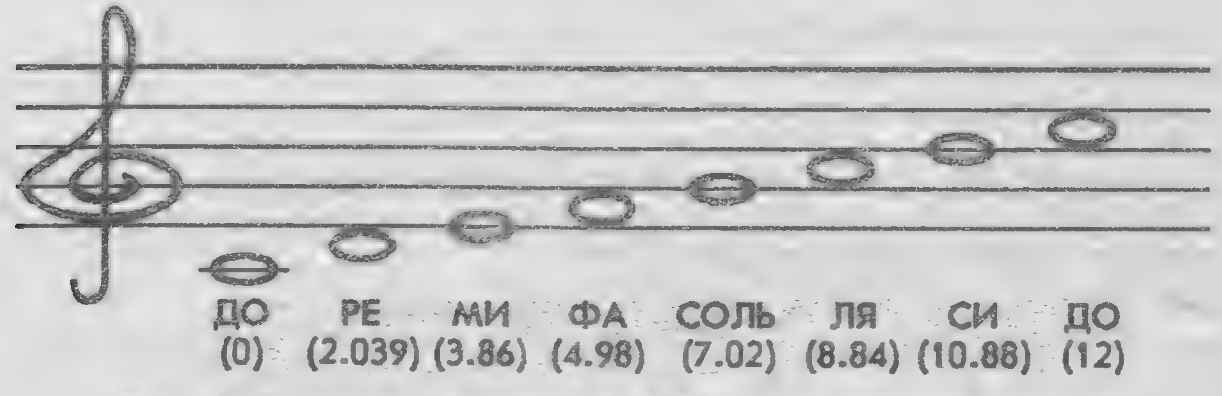


Рис. 4. Точные параметры BEEP для средней октавы.

ру, запишется фрагментик из популярной когда-то песни квартета ABBA «Money, money...»:

```
10 BEEP 0.15,7.02: BEEP 0.15,8.84: BEEP 0.15,10.88
20 BEEP 0.15,7.02: BEEP 0.15,10.88: BEEP 0.75,8.84
30 BEEP 0.15,8.84: BEEP 0.15,8.84: BEEP 0.15,10.88
40 BEEP 0.3,10.88: BEEP 0.6,7.02
```

Конечно, мы не стали бы занимать место под эту примитивнейшую программу, если бы не задались целью показать, как ее можно модифицировать с помощью какого-нибудь, еще не изученного нами средства.

Массивы

DIM

Первое, что приходит на ум при взгляде на длинную цепочку однотипных операторов BEEP, — это написать цикл типа:

```
10 FOR i=1 TO 11
20 BEEP T, H
30 NEXT i
```

Правда, в таком виде программка десять раз проиграет одну и ту же ноту (длительностью — T и высотой тона — H). Надо каким-то образом менять параметры BEEP при «прогоне» цикла, то есть необходимо организовать ряд (набор) переменных и уметь выбирать из этой последовательности нужную, указывая ее порядковый номер. Тогда можно будет в первую переменную одной из последовательностей (назовем ее T(1)) записать длительность звучания первой ноты, во вторую (T(2)) — длительность второй ноты и т. д. Во вторую последовательность H(i) занести одно за другим значения высот нот. Тогда цикл можно было бы записать следующим образом:

```
10 FOR i=1 TO 11
20 BEEP T(i), H(i)
30 NEXT i
```

Такие последовательности переменных, имеющих одно имя и отличающихся номером (*индексом*), называются *массивами*. Переменные, входящие в массив (*элементы массива* или *индексированные переменные*), можно применять как и обычные переменные. Только запись у них немного другая, да имя должно состоять только из одной буквы. И, конечно, у них есть огромное преимущество по сравнению с обычными переменными — их можно обрабатывать одним оператором в цикле, меняя индекс.

Перед применением массива его надо задать — определить имя и количество элементов в нем. Делается это так: пишется ключевое слово **DIM**, а за ним ставятся имя и, в скобках, *размерность* массива:

```
5 DIM T(11): DIM H(11)
```

Мы написали «размерность», поскольку массивы могут быть не только одномерными (с одним индексом), но и многомерными (с любым количеством индексов). В этом случае принято говорить о массивах с несколькими *измерениями*. Например, массив, задаваемый строкой DIM A(5, 10), двумерный. Он состоит из 50 (5×10) индексированных переменных. Для большей наглядности их можно мысленно расположить в виде таблицы с 5 строками и 10 столбцами:

```
A(1,1) A(1,2) ... A(1,10)
A(2,1) A(2,2) ... A(2,10)
... ... ...
A(5,1) A(5,2) ... A(5,10)
```

Обращаться с элементами этого массива можно так же, как и с элементами одномерного: указывай «координаты» переменной в таблице (номер строки и номер столбца) и работай, как с обычной переменной: присваивай значения с помощью LET, вставляй в выражения и т. д.

Для обработки всего массива потребуется уже не один, а два цикла — один должен быть «вложен» в другой. Вот, например, как будет выглядеть программа, присваивающая индексированным переменным значения, соответствующие их индексам, и распечатывающая массив на экране в виде таблицы:

```
10 DIM A(5,10)
20 FOR i=1 TO 5
30 FOR j=1 T0 10
40 LET A(i,j)=10*i+j: PRINT AT i*2, j*3; A(i,j)
50 NEXT j
50 NEXT i
```

Массивы могут быть составлены и из символьных переменных. Массив, заданный строкой DIM S$(10, 5), может рассматриваться либо как одномерный массив, состоящий из 10 символьных переменных длиной по 5 символов каждая: S$(1)...S$(10), либо как двумерный массив из 50 переменных длиной в один символ: S$(1,1)...S$(10,5).

Однако мы увлеклись описанием различных массивов. Пора вернуться к нашей музыкальной программке. Кстати, ведь ее можно и модернизировать: вместо двух одномерных массивов T() и H() организовать один двумерный. Это напрашивается само собой, ведь по отдельности эти два массива не имеют смысла:

```
 5 DIM M(11, 2)
...
10 FOR i=1 TO 11
20 BEEP M(i, 1), M(i, 2)
30 NEXT i
```

Все это хорошо, но только при условии, что каким-то образом элементам массива будут присвоены значения длительности и высоты нот. Если пойти прямым путем и написать 20 операторов

LET, то мы не только ничего не выгадаем, но, наоборот, сильно увеличим длину программы. Представьте себе конструкцию:

```
LET M(1,1)=1: LET M(1,2)=2: LET M(2,1)=3
```

и т. д. до

```
LET M(11,2)=22
```

Проще было бы оставить 11 операторов BEEP.

Данные в программе

DATA, READ, RESTORE

Разработчики Бейсика позаботились о нас и включили в него удобное средство для хранения и загрузки данных в переменные. Не тех данных, которые вводит в программу пользователь с помощью оператора INPUT, а данных, закладываемых программистом при ее написании. Например, параметров для операторов BEEP.

Делается это просто: пишется ключевое слово **DATA**, а за ним через запятую в нужном порядке размещаются данные. Параметры нашей мелодии (длительности и высоты нот) запишутся так:

```
200 DATA 0.15, 7.02, 0.15, 8.84, 0.15, 10.88, 0.15, 7.02, 0.15, 10.88,
    0.75, 8.84, 0.15, 8.84, 0.15, 8.84, 0.15, 10.88, 0.3, 10.88, 0.6, 7.02
```

Данные можно разместить как за одним оператором DATA, так и распределить между несколькими, то есть разбить строку 200 на две, три и более строк:

```
200 DATA 0.15, 7.02, 0.15, 8.84, 0.15, 10.88, 0.15, 7.02, 0.15, 10.88,
    0.75, 8.84
210 DATA 0.15, 8.84, 0.15, 8.84, 0.15, 10.88, 0.3, 10.88, 0.6, 7.02
```

Оператор DATA можно располагать в любом месте программы. Компьютер игнорирует его и переходит к выполнению следующей строки. Чаще всего данные помещают за основным текстом программы, чтобы не затруднять ее чтение.

Как только компьютер встретит оператор **READ** с одной или более (через запятую) стоящими за ним переменными, он моментально отыщет строку с DATA и загрузит в эти переменные данные из DATA. Например:

```
READ T, H: PRINT T;"  "; H
0.15  7.02
```

Первый попавшийся по тексту оператор READ заносит в переменные порцию данных из первого встреченного в программе оператора DATA. Следующие операторы READ продолжают последовательное считывание. То есть, повторив предыдущую строку, мы получим уже другой результат:

```
READ T, H: PRINT T;"  "; H
0.15  8.84
```

Следует особо следить за тем, чтобы количество считываний не превышало количество перечисленных за операторами DATA данных. Иначе на экране появится сообщение об ошибке: End of data.

Поскольку значения, указываемые за операторами DATA, могут быть как числовыми, так и символьными, то необходимо также следить за порядком их занесения в переменные. Числовые данные должны считываться в числовые переменные, символьные — в символьные.

Теперь, используя оператор READ, мы можем изящно загрузить параметры мелодии из DATA в массив M():

```
10 FOR i=1 TO 11
20 FOR j=1 TO 2
30 READ M(i, j)
40 NEXT j
50 NEXT i
```

Правда, получилось, что в компьютере параметры нашей мелодии хранятся в двух местах: и в тексте самой программы (в строке за DATA), и в массиве. Это нерационально, и от последнего, наверное, придется отказаться*.

Используя способность READ последовательно считывать данные, можно обойтись лишь двумя переменными. Надо только совместить процесс считывания и проигрывания:

```
300 FOR i=1 TO 11
310 READ T, H
320 BEEP T, H
330 NEXT i
```

Теперь эти строки можно использовать в программе.

А если захочется, чтобы мелодия звучала в нескольких местах программы? Переписать еще раз цикл с READ и BEEP, конечно, не представит труда, но как быть с данными? Ведь мы их уже считали. Не дублировать же и их?

Специально для того, чтобы иметь возможность многократно считывать информацию, записанную в DATA, в Бейсик введен оператор **RESTORE.** Он указывает компьютеру, что следующему за ним оператору READ необходимо считывать данные опять из первого DATA.

Поставив после RESTORE номер строки, в которой расположен какой-либо промежуточный оператор DATA, можно повторное считывание начать с него. Таким образом, в нашем примере можно организовать проигрывание музыкальной фразы с любого места. Надо только не забыть изменить конечное значение управляющей переменной цикла:

```
350 RESTORE 210
360 FOR i=1 TO 5
370 READ T, H
380 BEEP T, H
390 NEXT i
```

---

* Хотя и не следовало бы, учитывая возможность записи и загрузки массивов на магнитную ленту отдельно от основной программы, что бывает очень удобно. К этому вопросу мы еще вернемся в разделе, посвященном записи и загрузке информации (см. стр. 49).

Подпрограммы

GO SUB, RETURN

Можно еще как-то смириться с повтором фрагмента программы длиной в несколько строчек, но довольно часто приходится использовать по многу раз программные блоки, состоящие, скажем, из десятков строк. Кому не лень, пусть набирает эти строки несколько раз, но уважающий себя программист в такой ситуации, не задумываясь, организует повторение программного фрагмента как вызов *подпрограммы*. В то место программы, где необходимо использовать ранее написанный блок, он вставит строку

```
7 GO SUB 290
```

— что означает: перейти к подпрограмме, расположенной, начиная со строки 290. То же самое сделал бы и оператор GO TO, но с существенной оговоркой: после выполнения подпрограммы компьютер должен вернуться к оператору, следующему за **GO SUB**. Дать понять компьютеру, что подпрограмма закончена и пора возвращаться, необходимо с помощью оператора **RETURN**:

```
280 REM Подпрограмма для проигрывания мелодии
290 RESTORE
300 FOR i=1 TO 10
310 READ T, H
320 BEEP T, H
330 NEXT i
340 RETURN
```

Вот так будет выглядеть музыкальная подпрограмма. Вызов же ее мы уже оформили (GO SUB 290).

Эту подпрограмму можно сделать более универсальной, введя в нее две переменные: NRest — номер строки оператора DATA, начиная с которого нужно считывать данные для проигрывания фрагмента мелодии, и ColB — количество нот в фрагменте. Сама подпрограмма и ее вызов будут выглядеть так:

```
 6 LET NRest=200: LET ColB=11
 7 GO SUB 280
...
77 LET NRest=210: LET ColB=5: GO SUB 280
...
```



```
280 REM Подпрограмма проигрывания мелодии и ее фрагментов
290 RESTORE NRest
300 FOR i=1 TO ColB
310 READ T, H
320 BEEP T, H
330 IF INKEY$<>"" THEN GO TO 350
340 NEXT i
350 RETURN
```

В 330 строку мы включили процедуру выхода из подпрограммы при удержании любой клавиши. Пользователь должен иметь возможность в любой момент прервать мелодию и приступить (вернуться) к работе с программой.

При написании сложной по структуре программы бывает полезно оформлять подпрограммами и неповторяющиеся ее части. Такую программу легче читать и отлаживать.

РАБОТА С МАГНИТОФОНОМ

Запись программ

SAVE

Написанная программа будет сохраняться в памяти компьютера до тех пор, пока мы его не выключим. Но для нас важно, чтобы ею можно было пользоваться и завтра, и послезавтра. Должен быть некий способ сохранения программ с возможностью в последующем быстрой загрузки их в память компьютера. Самый простой способ — запись на магнитную ленту обычного магнитофона (как кассетного, так и катушечного). То, что создается на магнитной ленте при записи туда программы, называется *файлом*.

Подключим магнитофон к специальному гнезду компьютера, наберем ключевое слово **SAVE**, а вслед за ним заключенное в кавычки имя файла с программой. Имя придумаем сами, помня лишь о том, что оно не должно содержать более десяти символов, но и менее одного. Например:

SAVE "DOLLAR"

На ввод этого оператора компьютер откликнется сообщением:

Start tape, then press any key.

– то есть попросит включить магнитофон на запись* и нажать любую клавишу на компьютере. После этого компьютер начнет «шипеть», а по бордюру побегут цветные полоски — значит, информация записывается.

По окончании записи появится стандартное сообщение:

0 OK

---

* Обеспечение надежной записи и считывания информации с магнитной ленты — это тема отдельного разговора. Здесь же посоветуем только устанавливать уровень записи почти на максимум.

Можно записать программу на магнитофон таким образом, чтобы после загрузки для ее запуска не требовалось выполнения оператора RUN. А в этом действительно есть необходимость. Ведь наша программа должна быть рассчитана на непросвещенного пользователя, от которого грех требовать выполнения даже такого элементарного действия, как запуск программы.

Для обеспечения автоматического старта при загрузке программы (*автостарта*) к оператору SAVE нужно добавить ключевое слово LINE с указанием строки старта:

```
SAVE "DOLLAR" LINE 1
```

Мы задали автостарт с первой строки, чем обеспечили выполнение программы с самого начала. Подставив другой номер, можно задать автостарт с любой строки программы.

Проверка правильности записи программы

VERIFY



Бейсик ZX Spectrum позволяет после записи программы на магнитофон убедиться в безошибочности проведения этой операции. Перемотаем магнитную ленту в начало записанного нами файла, наберем оператор

```
VERIFY "DOLLAR"
```

– включим магнитофон на воспроизведение и нажмем **Enter**. Когда компьютер встретит на ленте файл с указанным именем, он начнет сравнивать его с программой, записанной в его памяти. Если тексты программы в памяти и на ленте совпадают, то выдается сообщение 0 OK, увидев которое можно вздохнуть спокойно и выключить компьютер.

Если же появится надпись Tape loading error, значит, запись не удалась и нужно повторить ее еще раз*.

Загрузка программы

LOAD

Сохраненную на ленте программу (файл) загрузить в память компьютера после некоторой тренировки достаточно просто. Выполним оператор

```
LOAD "DOLLAR"
```

* Срыв записи может происходить вследствие дефектов ленты, недостаточного (редко – избыточного) уровня записи, плохой настройки магнитной головки, неисправности соответствующих узлов компьютера или магнитофона и т. д.

– не забыв предварительно включить магнитофон на воспроизведение. Если, просматривая ленту, компьютер встретит файл с указанным именем, то он без лишних вопросов загрузит его к себе в память. При успешном завершении процесса на экране появится соответствующее сообщение или загруженная программа будет запущена (если она была сохранена оператором SAVE LINE).



Когда имя программы точно не известно, его можно и не указывать, заменив «пустой строкой» — двумя кавычками без текста:

```
LOAD ""
```

Выполняя этот оператор, компьютер загрузит первую встретившуюся на ленте бейсик-программу.

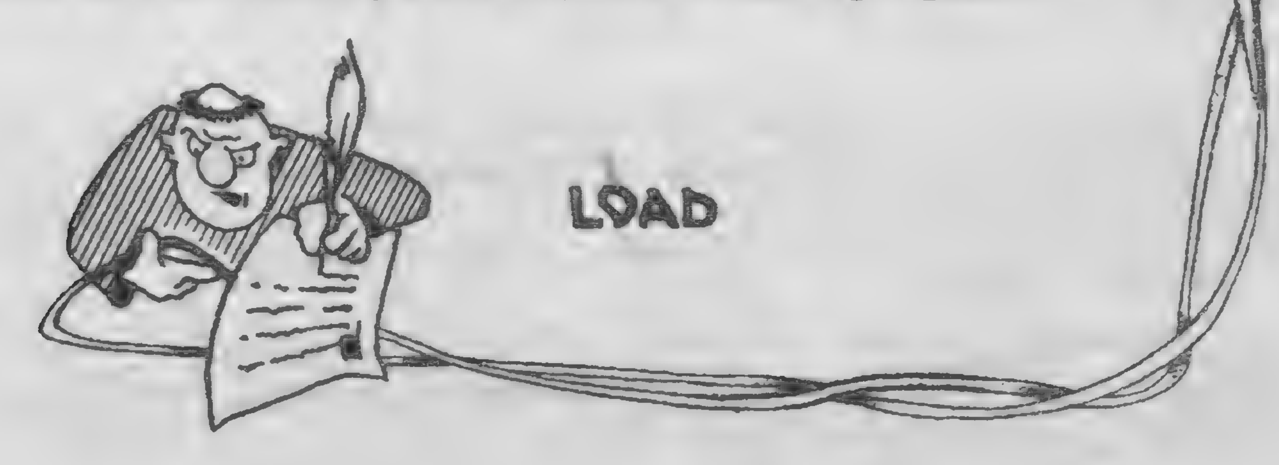
Подгрузка программ

MERGE

После загрузки новой программы ранее находившаяся в памяти компьютера программа бесследно исчезает. Но можно загрузить вторую программу и не тронув старой, из двух программ сделать одну. Если в памяти компьютера находится наша программа "DOLLAR", а на ленте имеется программа "PROG", то после ее подгрузки оператором

```
MERGE "PROG"
```

в листинге можно обнаружить строки обеих программ.



При слиянии программ следует особо следить за тем, чтобы в них не было строк с совпадающими номерами. Иначе строки ранее загруженной программы ("DOLLAR") будут попросту заменены на строки новой ("PROG") с совпадающими номерами. Та же участь ожидает и одноименные переменные.

Запись и загрузка массивов

SAVE...DATA, LOAD...DATA

В своей «валютной» программе мы отказались от использования массива для хранения параметров мелодии. Если бы мы этого не сделали, то нам мог бы пригодиться оператор SAVE с ключевым словом DATA. С его помощью можно сохранять массивы отдельно от программы, что позволяет существенно сократить ее размер. Ведь после того, как однажды при наладке программы рассчитаны или считаны из оператора DATA значения элементов массива и он сохранен на магнитной ленте, часть программы, делающая расчет или считывание, может быть со спокойной совестью удалена.

Более того, возможность сохранения массивов отдельно от программы позволяет, в зависимости от желания пользователя, варьировать ее функции, подгружая одни или другие файлы с массивами. Например, при загрузке нашей программы можно было бы попросить пользователя по своему усмотрению выбрать музыкальное сопровождение программы. Для этого следовало бы при написании программы подготовить и сохранить на магнитной ленте несколько массивов с параметрами различных мелодий.

Сам процесс записи массива ничем не отличается от записи всей программы, только после имени файла нужно записать ключевое слово DATA, а после него имя массива с двумя пустыми скобками:

```
SAVE "MELOD" DATA M()
```

Соответственно, считывание массива производится оператором

```
LOAD "MELOD" DATA M()
```

Правильность записи массива можно проверить оператором

```
VERIFY "MELOD" DATA M()
```

Запись и загрузка экранного изображения

SAVE...SCREEN$, LOAD...SCREEN$, VERIFY...SCREEN$

Сохранить на магнитной ленте можно и картинку, изображенную на экране, записав ее на ленту с помощью оператора

```
SAVE "PICTURE" SCREEN$
```

После чего, вставив в программу строку:

```
5 LOAD "PICTURE" SCREEN$
```

— можно (если не выключать магнитофон после загрузки программы) загрузить картинку с ленты прямо на экран. Для этого надо поместить на ленте файл с картинкой вслед за программным файлом.

Лучше написать специальный загрузочный модуль — коротенькую программку, которая должна загрузиться и запуститься первой и обеспечить последовательную загрузку сначала заставки (чтобы скрасить ожидание), потом основного текста программы.

В нашем случае программа-загрузчик может выглядеть так:

```
5 LOAD "PICTURE" SCREEN$
10 LOAD "DOLLAR"
```

Запись экранного изображения можно проверить на отсутствие ошибок оператором

```
VERIFY "PICTURE" SCREEN$
```

Запись и загрузка содержимого областей памяти

SAVE...CODE, LOAD...CODE, VERIFY...CODE

Сохранить на магнитной ленте экранное изображение можно и другим способом — как область памяти:

```
SAVE "PICTURE" CODE 16384, 6912
```

Приведенный оператор запишет на ленту содержимое области памяти, начиная с адреса* 16384 (первый параметр после ключевого слова CODE) длиной в 6912 байт** (второй параметр). Именно в этой области памяти хранится информация об экранном изображении.

Таким же образом, указывая начальный адрес и размер в байтах, можно скопировать на магнитную ленту любую область памяти компьютера. После чего загрузить ее обычным образом, только указав компьютеру, что следует ожидать блок кодов:

```
LOAD "PICTURE" CODE
```

Если после CODE не стоит никаких параметров, то блок кодов загрузится в то место, с которого его записывали. Но можно и изменить адрес начала области памяти, куда должна производиться загрузка блока, и указать количество загружаемых байт:

```
LOAD "PICTURE" CODE 16384, 6912
```

---

* Что такое адрес в памяти, см. на стр. 60.

** Напомним, байтом называется восьмиразрядное двоичное число.

Запись блока кодов можно проверить на отсутствие ошибок оператором

```
VERIFY "PICTURE" CODE 16384, 6912
```

РАБОТА С ПРИНТЕРОМ

LPRINT, LLIST

ZX Spectrum с подключенным к нему принтером открывает перед программистом новые горизонты.

Во-первых, намного упрощается отладка больших программ. Ведь с помощью оператора **LLIST** (аналогичного LIST) можно вывести листинг на принтер и, уже не уткнувшись в экран, а аккуратно разложив перед собой листки с текстом программы, разбираться: что, куда и как.

Во-вторых, имея принтер, результаты выполнения собственноручно написанных программ можно не только наблюдать на экране монитора, но и получать материальный продукт их работы в виде распечаток. Так, например, при работе с нашей «долларовой» программой можно автоматически распечатывать ведомость расчетов. Для этого потребуется лишь добавить в программу несколько строчек:

```
 1 LET NP=1
 2 INPUT "Введите сегодняшнее число "; LINE D$;
 3 LPRINT "Ведомость на "; D$
...
76 LPRINT NP;" " ; Rub; " — "; Dol; "$ ";
77 LET NP=NP+1
```

В переменной NP хранится номер очередного вычисления.

С помощью оператора COPY на принтере можно получить и целиком копию экрана. Причем безразлично, что в этот момент находится на экране — текст или графическая картинка.

При отключенном принтере операторы LLIST, LPRINT и COPY игнорируются. Печать всегда можно остановить нажатием **Break (CS/Space)**.

Как компьютер общается с принтером, на какие управляющие символы и ключевые слова тот реагирует, зависит от многих причин. В основном, от специальной подпрограммы (драйвера), обслуживающей принтер. Тут уж кто как устроится.

О ЧЕМ ЕЩЕ МОЖНО РАССКАЗАТЬ

В этом разделе мы дадим информацию, без которой наш рассказ был бы неполным.

Экспоненциальная форма записи чисел

Кроме обычной формы представления чисел Speccy понимает и экспоненциальную, записывая степень через букву E. Например, числа $5{,}5 \cdot 10^5$, $2{,}3 \cdot 10^{-1}$ запишутся следующим образом:

```
PRINT 5.5E5
550000

PRINT 2.3E-1
0.23
```

Оператор PRINT при расчетах выводит результат не более чем с 8 значащими цифрами. Причем и те не всегда точны, даже при обращении с целыми числами. Например, выполним:

```
PRINT 1E10+1-1E10: PRINT 1E10-1E10+1
```



Сравнение чисел

Сравнение чисел может использоваться не только как условие в операторе IF...THEN, но и при обычных математических расчетах. Например, попросим PRINT вывести результат выражений, в которых сравниваются два числа:

```
PRINT 5<4
0

PRINT 5>4
1
```

Что и следовало ожидать: результат истинного сравнения — 1, ложного — 0. В выражениях, сравнивающих величины, используются все возможные сочетания знаков: «равно» (=), «больше» (>), «меньше» (<), «больше или равно» (>=), «неравно» (<>), «меньше или равно» (<=).

Выражения со сравнением можно вставлять и в «нормальные» числовые выражения:

```
PRINT 5+2*(3>6)
5
```

Сравнив числа 3 и 6, компьютер сделал заключение, что выражение в скобках ложно и подставил вместо него ноль.

Сравнение символьных значений

Поскольку каждому символу соответствует некий числовой эквивалент — код, то можно сравнивать не только числовые, но и символьные значения. Например, казалось бы, абсолютно бессмысленное с точки зрения математики выражение:

```
PRINT "A"<"B"
1
```

— для компьютера имеет вполне конкретное значение. Подумав, он сделал заключение, что оно верно, поскольку код буквы **A** меньше кода буквы **B**. При сравнении символьных значений, состоящих более чем из одного символа, происходит последовательное сравнение кодов до тех пор, пока не встретятся неравные коды:

```
PRINT "AAAAC">"AAAAB"
1
```

При прочих равных условиях меньшим будет признано более короткое символьное значение:

```
PRINT "AAAA">"AAAAB"
0
```

Двоичное счисление

BIN

Реально компьютер обрабатывает числа, записанные в двоичном счислении: нулями и единицами. Поэтому часто приходится сталкиваться с необходимостью преобразовывать числа, записанные в двоичном виде, в привычные и для глаза, и для счета, десятичные. Специализируется на этом функция **BIN**. Например:

```
PRINT BIN 0
0
PRINT BIN 1
1
PRINT BIN 10
2
PRINT BIN 11
3
PRINT BIN 100
4
```

Мы привели так много примеров, чтобы хоть как-то помочь тем, кто впервые сталкивается с двоичной записью чисел.

Математически оператор BIN эквивалентен формуле:

$$D_0 \times 2^0 + D_1 \times 2^1 + D_2 \times 2^2 + D_3 \times 2^3 + \ldots$$

– где D_0, D_1, D_2, D_3, соответственно, разряды двоичного числа, читаемые справа налево и принимающие значения 0 или 1. Переведем из двоичного в десятичный вид число 1101:

$D_0=1$, $D_1=0$, $D_2=1$, $D_3=0$, $D_4=1$

$$1\times2^0 + 0\times2^1 + 1\times2^2 + 1\times2^3 = 13$$

Проверим:

```
PRINT BIN 1101
13
```

Так оно и есть.

Случайные числа

RND, RANDOMIZE

В набор числовых функций Бейсика входит функция **RND** — генератор псевдослучайных чисел. Потребность в получении случайных чисел возникает, например, при написании игровых программ, обучающих систем (выдача неповторяющихся заданий) и во многих других случаях.

Для RND не нужен аргумент — при вызове функция возвращает случайное число в интервале от 0 до 1. Распечатаем на экране много-много раз значения, возвращаемые RND:

```
10 FOR i=1 TO 100
20 PRINT RND; " ";
30 NEXT i
```



Получим ряд чисел, между которыми нет никакой видимой зависимости, что и является признаком случайности. Однако если генерировать числа с помощью RND долго-долго, то их последовательность начнет повторяться. Поэтому в название функции добавлено слово «псевдо». Хотя в подавляющем большинстве случаев этим «псевдо» можно пренебречь.

Существенным неудобством RND является то, что после запуска (или сброса) компьютера функция каждый раз начинает генерировать с одного и того же числа одну и ту же последовательность (происки все того же «псевдо»). Пришлось ввести в Бейсик специальный оператор **RANDOMIZE**. Выполняя его с указанием вслед за ним произвольного числа в интервале 1...65535, можно уста-

навливать начало последовательности случайных чисел. Если требуется совершенно случайная последовательность с совершенно случайным началом, то нужно вслед за RANDOMIZE поставить ноль либо вообще ничего не писать. Выполним несколько раз (каждый раз сбрасывая компьютер) строку:

```
RANDOMIZE: PRINT RND
```

Теперь-то уж мы приблизились к случайности — функция RND каждый раз выбрасывает различные значения.

Для генерации случайных чисел в интервалах, отличных от стандартно заданного для функции RND (0...1), используются элементарные алгебраические выражения. Например, оператор

```
PRINT 2+3*RND
```

будет выдавать случайные числа в интервале 2...5. Известная нам функция INT поможет генерировать целые случайные числа. Скажем, с помощью оператора

```
PRINT 1+INT(RND*9)
```

можно получить последовательность целых случайных чисел от 1 до 10.

Анализ атрибутов знакоместа

ATTR

При написании игровых программ, да и во многих других случаях,часто бывает необходимо определить атрибуты знакоместа экрана, то есть выяснить, какими цветами оно раскрашено. В Spectrum-Бейсике этим занимается функция **ATTR**. Аргументами для нее являются координаты тестируемого знакоместа, размещаемые за ней в скобках через запятую: номер строки (0...22) и номер столбца (0...31). В качестве результата ATTR возвращает числовое значение от 0 до 255, которым зашифрованы значения атрибутов знакоместа.

Выполним программку:

```
10 PRINT AT 10,10; INK 1; PAPER 6; BRIGHT 1; FLASH 1;"*";
20 PRINT ATTR(10, 10)
RUN
*241
```

В ней задаются, а затем считываются атрибуты знакоместа с координатами (10, 10). Число 241 получается в результате вычисления выражения, в котором <INK>, <PAPER>, <BRIGHT> и <FLASH> — значения атрибутов:

1×<INK>+8×<PAPER>+64×<BRIGHT>+128×<FLASH>

— то есть в нашем случае:

$$1\times1+8\times6+64\times1+128\times1=241$$

Символы псевдографики

В ZX Spectrum коды от 32 до 127 (см. табл. 1 на стр. 100) соответствуют стандартному набору компьютерных символов (кроме знаков £ и ©). Этот набор называется ASCII-коды (American Standard Code for Information Interchange).

Остальные символы не входят в ASCII. Символы с кодами от 128 до 143 включительно — это так называемые *псевдографические символы*. «Графические» они потому, что представляют из себя квадратики, прямоугольники и т. п. Их изображение помещено на цифровых клавишах (от 1 до 8). «Псевдо» — потому, что по сути они такие же символы, как и буквы, то есть выводятся на экран нажатием соответствующих клавиш.

Переход в режим псевдографики происходит после одновременного нажатия двух клавиш: **CS/9** (**Graphics**). Курсор меняется на **[G]**. Если одновременно с нажатием цифровых клавиш удерживать клавишу **SS**, графические символы будут выводиться в инверсном виде. Для выхода из режима **[G]** нужно нажать клавиши **CS/9** или даже просто **9**.

Символы, определяемые пользователем

В режиме псевдографики, кроме нанесенных на цифровые клавиши графических символов, на экран выводятся и так называемые символы, определяемые пользователем (UDG — User Defined Graphics). Их может быть до 22, и «привязываются» они к клавишам от **A** до **U**. Это могут быть и дополнительные графические символы, и произвольные значки, и — что часто можно встретить в отечественных Speccy — буквы русского алфавита. Этим новым символам присваиваются коды от 144 до 155.

Наша фантазия при создании новых символов ограничена лишь размерами знакоместа (8×8 пикселей). Попробуем «сконструировать» значок, показанный на рис. 5.

Для записи изображения символа в память компьютера его надо представить последовательностью из 8 чисел, соответствующих строкам пикселей знакоместа. Каждая строка представляется в виде двоичного числа, исходя из принципа: есть точка в данной позиции строки — ставится 1, нет — 0:

```
00011100
00011100
00001000
01111111
00011100
00011100
00010100
00010100
```

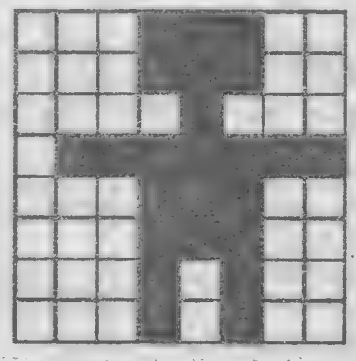


Рис. 5. Эскиз символа UDG.

Эти числа, преобразованные с помощью функции BIN в десятичные, последовательно записываются в память компьютера, начиная с адреса, возвращенного функцией USR*. Аргументом USR должен быть символ клавиши, за которой мы хотели бы «закрепить» новый значок. Например, если мы запишем последовательность полученных чисел, начиная с адреса USR "A", то в режиме курсора [G] при нажатии на клавишу A будет выводиться наш символ.

Запись в память осуществляет специальный оператор Spectrum-Бейсика POKE**, вслед за которым через запятую нужно указать, куда записывать (адрес) и что записывать:

```
POKE USR "A", BIN 00111100
```

Второе число записывается по следующему адресу:

```
POKE USR "A"+1, BIN 01000010
```

И так восемь раз. После этого можно перейти в режим курсора [G] и попробовать выводить значок, нажимая клавишу A***.

Управление выводом на экран с помощью «знаков препинания»

В какое место на экране будет размещать текст оператор PRINT, зависит от знака, стоящего в конце предыдущего PRINT. Если он оканчивался точкой с запятой (;), то, как мы уже знаем, текст будет выводиться на экран, начиная со следующего знакоместа в той же строке:

```
10 PRINT "BASIC";
20 PRINT " IS MY LOVE"
RUN
BASIC IS MY LOVE
```

Запятая (,) укажет на то, что вывод нужно продолжить с середины строки (если, конечно, последний помещенный символ уже не перевалил за середину; в таком случае данные будут размещены с начала следующей строки):

```
10 PRINT "SPECCY",
20 PRINT " IS MY LOVE"
RUN
SPECCY IS MY LOVE
```

Если вообще какой-либо знак отсутствует, значит, следующему оператору PRINT следует выводить символы с новой строки, то есть перед выводом будет выполнена операция «перевод строки».

---

* Функция USR используется также для вызова подпрограмм в машинных кодах (см. «Справочник»).

** С ним мы более подробно познакомимся позже.

*** Более подробно о формировании символов UDG читайте в [1] (в квадратных скобках мы будем указывать порядковый номер издания в списке литературы, приведенном в конце книги).

PRINT сам по себе (без данных) вставляет «пустую строку» и переводит текущую позицию вывода в начало следующей.

В операторе PRINT может использоваться и апостроф ('). Натолкнувшись на него, компьютер осуществит перевод строки.

Знаки препинания могут отделять различные блоки данных, выводимых одним оператором PRINT:

```
10 PRINT 1234; " QWERT", "ASDF" ' "ZXCV"
1234 QWER              ASDF
EZXCV
```

Управляющие символы

Символы с кодами от 0 до 31 (см. табл. 2), вообще говоря, символами не являются: у них нет собственного, отображаемого на экране начертания. Это так называемые *управляющие символы*.

Управляют они, например, выводом на экран компьютера. Так, попытка вывести на экран управляющий символ с кодом 6 приведет к результату, аналогичному действию запятой, поставленной между данными после PRINT:

```
PRINT "KU"; CHR$ 6; "KU"
KU                     KU
```

Управляющий символ с кодом 13 действует аналогично апострофу: указывает, что вывод нужно продолжить с начала следующей строки.

Символы CHR$ 22 и CHR$ 23 в операторе PRINT выполняют те же функции, что и AT и TAB. После этих управляющих символов указывается по два параметра (второй параметр для CHR$ 23 игнорируется). Оператор

```
PRINT CHR$ 22+CHR$ 1+CHR$ 10; 5; CHR$ 23+CHR$ 11+ CHR$ 0; 5
```

аналогичен оператору

```
PRINT AT 1, 10; 5; TAB 11; 5
```

Управляющий символ CHR$ 8 называется «забой» (backspace). Он переводит текущую позицию вывода на одно знакоместо назад. То есть символ, следующий за «забоем», оператор PRINT выводит на место последнего, напечатанного до него:

```
10 PRINT "ABA"; CHR$ 8; "C"
RUN
ABC
```

Символы с кодами 16...21 управляют атрибутами экрана и могут использоваться в операторе PRINT вместо соответствующих ключевых слов:

```
INK     — CHR$ 16
PAPER   — CHR$ 17
FLASH   — CHR$ 18
BRIGHT  — CHR$ 19
INVERSE — CHR$ 20
OVER    — CHR$ 21
```

Например, оператор

```
PRINT CHR$ 17+CHR$ 6;"ABC"
```

равнозначен

```
PRINT PAPER 6;"ABC"
```

Управление цветом в режиме курсора [E]

Назначить цвет фона и цвет тона можно еще одним способом: раскрашивая предназначенный для вывода текст непосредственно при его наборе. Нужно только, когда потребуется изменить цвет, войти в режим курсора [E] и нажать цифровую клавишу, соответствующую номеру желаемого цвета. Нажатие цифровой клавиши совместно с **CS** задает цвет фона, без **CS** — цвет тона. При этом в текст будут вставляться «невидимые» управляющие символы CHR$ 17 и CHR$ 16.

В режиме курсора [E] можно управлять и другими атрибутами экрана:

8 — BRIGHT 0
9 — BRIGHT 1
CS/8 — FLASH 0
CS/9 — FLASH 1

Инверсный вывод можно задать при наборе текста непосредственно в режиме курсора **[L]**:

CS/3 — INVERSE 0
CS/4 — INVERSE 1

Все действия по раскраске, выполненные в режиме курсора [E], можно легко отменить. «Невидимые» коды управления цветом удаляются двойным нажатием **Delete** (**CS/0**).

СИСТЕМНЫЕ ПЕРЕМЕННЫЕ

Интерпретатору Бейсика для «служебного пользования» в компьютере отводится специальная область оперативной памяти. В ячейках этой области хранятся настроечные параметры интерпретатора, а также данные, необходимые для обработки бейсик-программ. Читая содержимое этих ячеек, можно узнать много интересного о состоянии компьютера. Записывая же в них необходимые нам значения, можно влиять на работу компьютера.

Эти ячейки памяти называют *системными переменными*. У каждой из переменных есть свое имя и фиксированное положение в памяти*.

---

* Наверное, нет необходимости пояснять, что системные переменные ничего общего не имеют с переменными бейсик-программ.

Многие из системных переменных непосредственного отношения к Бейсику не имеют, но использование некоторых из них намного расширяет возможности программиста, тем более, что изменять значения системных переменных можно с помощью операторов Бейсика.

Чтение содержимого памяти

PEEK

Память ZX Spectrum состоит из множества отдельных ячеек, в каждой из которых может быть записано число от 0 до 255 (байт). Всего ячеек 65535. Для того чтобы различать ячейки, каждой из них приведено в соответствие число от 0 до 65535, называемое *адресом*. С помощью оператора **PEEK** прямо из Бейсика можно читать содержимое любых ячеек памяти. Выполним в непосредственном режиме оператор

```
PRINT PEEK 23560
13
```

По сути, мы выяснили, что в ячейке памяти с адресом 23560 записано число 13. «Зациклим» считывание этой ячейки:

```
10 PRINT PEEK 23560
20 GO TO 10
RUN
13
```

Вполне естественно, что первоначальный результат выполнения программы будет таким же, как и при выполнении строки. Но если теперь, не останавливая программу, нажимать разные клавиши, на экране будут появляться коды соответствующих символов.

Итак, мы познакомились с системной переменной **LAST_K** (адрес 23560), в которой постоянно хранится код последней нажатой клавиши.

В некоторых случаях пользоваться системной переменной LAST_K предпочтительнее, чем функцией INKEY$. Ведь INKEY$ возвращает символ клавиши, нажатой именно в момент ее выполнения, и требует обязательного «зацикливания» программы на опросе клавиатуры. Переменная LAST_K сохраняет код клавиши и после ее отпускания. Это позволяет после завершения программой каких-либо действий проверить, была ли нажата клавиша в момент выполнения этих действий.

Системный счетчик

Системный счетчик — внутренние часы ZX Spectrum. Под него в памяти компьютера отведено три ячейки памяти с адресами

23672/73/74*. Они носят общее название «системная переменная **FRAMES**».

После включения или сброса компьютера содержимое всех трех ячеек обнуляется. По прошествии 1/50 секунды в ячейку 23762 записывается число 1. И далее, через каждые 1/50 секунды ее содержимое увеличивается на 1. При достижении значения 255 ячейка 23762 обнуляется, а содержимое ячейки 23673 увеличивается на 1. И, соответственно, когда переполняется ячейка 23673 (записанное в ней значение достигает 255), к содержимому ячейки 23674 прибавляется 1. Следовательно, в нашем распоряжении имеются часы, отсчитывающие время с момента включения компьютера с точностью в 1/50 секунды непрерывно в течение

$$(255+255\times256+255\times256\times256)/50=335544{,}3 \text{ секунд,}$$

то есть 3 суток и еще 21 часа. Потом все сначала.

Для определения числа полных секунд, прошедших с момента включения или сброса компьютера, нужно рассчитать следующее выражение:

```
LET T=INT((65536*PEEK 23674+256*PEEK 23673+PEEK 23672)/50)
```

Таким образом можно смоделировать на компьютере часы:

```
10 INPUT "Час ";hour
20 INPUT "Минуты ";min
30 POKE 23674,0: POKE 23673,0: POKE 23672,0
40 LET t=PEEK 23672+2569*PEEK 23673+65536*PEEK 23674+
   hour*180000+min*3000
50 LET h=INT (t/180000)
60 LET h$=STR$ h
70 IF LEN h$=1 THEN LET h$="0"+h$
80 LET h$=h$( TO 2)+":"
90 LET m=INT ((t-h*180000)/3000)
100 LET m$=STR$ m
110 IF LEN m$=1 THEN LET m$="0"+m$
120 LET m$=m$( TO 2)+":"
130 LET s=INT ((t-h*180000-m*3000)/50)
140 LET s$=STR$ s
150 IF LEN s$=1 THEN LET s$="0"+s$
160 LET s$=s$( TO 2)
170 PRINT AT 20,24;h$;m$;s$
180 G0 TO 40
```

Эти «часы» можно включить в любую программу. Следует только учитывать, что на время ожидания ввода данных операторами INPUT вывод времени на экран будет остановлен**. Чтобы задержка была не так заметна, лучше из программы, моделирующей часы,

---

* В таком виде мы будем приводить адреса системных переменных, занимающих более одной ячейки. В данном случае это адреса 23672, 23673, 23674.

** Это относится и к другим операторам, приостанавливающим выполнение программы (**LOAD, SAVE, PAUSE**).

исключить строки 130...160 и убрать переменную s$ из строки 170, то есть не выводить на экран секунды.

Запись в память

POKE

Запись чисел в память производится с помощью оператора **POKE**. Вслед за ним через запятую располагаются два значения: адрес от 0 до 65535 и заносимое в память по указанному адресу число от 0 до 255.

Невидимая точка на экране

Оператор DRAW начинает чертить отрезок линии или дуги от последней выставленной на экране точки. Координаты этой точки он берет из системной переменной **COORDS**. В ячейках с адресами 23677 и 23678 этой переменной хранятся, соответственно, значения координат *x* и *y* последней выставленной на экран точки. Зная это, можно, не выставляя точки с помощью оператора PLOT, записывать начальную координату для DRAW сразу в переменную COORDS:

```
10 POKE 23677,100: POKE 23678, 100
20 DRAW 50, 50
```

Другое полезное применение в программе переменной COORDS — это получение из нее с помощью PEEK информации о координатах последней выставленной точки.

Окраска экрана

Оперируя системными переменными, можно добиться того, что невозможно сделать с помощью обычных операторов и функций Бейсика.

Например, в Бейсике нет оператора, изменяющего все атрибуты служебного экрана (оператор BORDER устанавливает только цвет фона). Задать их можно только, записывая соответствующие значения в системную переменную **BORDCR** (адрес 23624). Число, которое необходимо записать по этому адресу, формируется по принципу, изложенному в описании функции ATTR (см. стр. 55).

Определяя через системную переменную BORDCR цвет фона служебного экрана, мы одновременно задаем цвет бордюра.

Точно также, выполнив лишь один оператор

```
POKE 23693, X
```

– можно задать атрибуты и основного экрана (вместо привычных INK, PAPER, FLASH и BRIGHT). 23693 — это адрес системной переменной **ATTR_P**, значение которой определяет «окрас» основного экрана. X — число, которым зашифрованы атрибуты. Может, этот способ сложнее, но он короче и изящнее.

Другие операции с экраном

Оперируя системными переменными, можно менять привычные характеристики Бейсика ZX Spectrum. Например, можно увеличить основной экран: занять под него все 24 строки. Для этого нужно модифицировать содержимое системной переменной **DF_SZ** (адрес 23659), задающей количество строк в служебном экране:

```
POKE 23659,0
```

Правда, в этом случае следует исключить в программе вывод на служебный экран, иначе не оберешься беды. Когда же без служебного экрана не обойтись (при работе с оператором INPUT или при ожидании вывода сообщения, например, о завершении работы программы) необходимо восстановить содержимое DF_SZ:

```
POKE 23659,2
```

Существует системная переменная **SCR_CT** (адрес 23692), задающая количество строк текста, передвигаемых вверх по экрану без запроса scroll?. Требуемое значение этой переменной устанавливается только из программы (изменить SCR_CT в непосредственном режиме невозможно).

Настройка клавиатуры

Меняя значения системных переменных **REPDEL** (адрес 23561) и **REPPER** (23562), можно по своему вкусу настроить клавиатуру компьютера. В переменной REPDEL хранится число, определяющее величину задержки (в 1/50 сек) между нажатием клавиши и началом автоматического повторения нажатия (*автоповтора*). REPPER задает период автоповтора (также в 1/50 сек).

При желании можно изменить длительность звучания сигнала, возникающего при нажатии клавиш. Число, пропорциональное времени его звучания, содержится в системной переменной PIP (23609).

Звуковой сигнал возникает и при переполнении буфера редактора и некоторых других ошибках. Длительность сигнала регулируется значением системной переменной **RASP** (23608).

Полезная информация о памяти

Например, нас может заинтересовать длина написанной бейсик-программы. Выберем из системных переменных **PROG** (23635/36) и **VARS** (23627/28) адреса начала бейсик-программы и начала области бейсик-переменных и вычтем одно из другого:

```
PRINT (PEEK 23635+256*PEEK 23635)—(PEEK 23627+256*PEEK 23628)
```

Под системные переменные, хранящие значения адресов, отводятся по две ячейки памяти, поскольку в одну ячейку можно записать число не более 255. Для записи числа до 65535 (максимальный адрес памяти ZX Spectrum) его разбивают на два числа. Сначала делят на 255, после чего остаток от деления заносят в первую ячейку (младший байт), а целую часть — во вторую (старший байт). Например, число 64768 разобьется на два байта так:

```
LET L=64768-256*INT(64768/256): REM Младший байт
LET H=INT(64768/256): REM Старший байт
```

Умение записывать значения адресов в системные переменные может пригодиться, например, при оперировании с символами, определяемыми пользователем (см. стр. 56). Адрес начала области памяти, в которой размещается информация об этих символах, хранится в системной переменной **UDG** (адреса 23675/76). Изменяя ее содержимое, можно определить несколько наборов символов и по необходимости в программе переходить с одного на другой.

Аналогично можно переключать и основной набор символов. Стандартный набор символов ZX Spectrum «зашит» в постоянном запоминающем устройстве (ПЗУ), начиная с адреса 15616. Под хранение этого адреса выделена системная переменная **CHARS** (адреса 23606/07). Зная это, мы можем создать в памяти (или загрузить с магнитофона как кодовый блок) новый набор символов и, переписав значение CHARS, переключить на него работу компьютера. Как создавать символы, мы описывали в разделе, посвященном псевдографике. Переопределяя адрес начала текущего набора символов, необходимо помнить, что первые 32 кода (от 0 до 31) являются управляющими и им не соответствуют никакие символы. Поэтому адрес начала размещения символов больше числа, записанного в CHARS, на 256 (32×8)*. Следовательно, загрузив новый набор символов по адресу 64768, для подключения его в CHARS нужно записать число 64512 (64768–256)**.

Мы рассказали далеко не о всех системных переменных. Многим из неупомянутых трудно придумать какое-либо практическое применение в бейсик-программах. Для понимания назначения других недостаточно знаний, которые можно почерпнуть из этой книги. Полный список системных переменных и их описание можно найти в [1].

СЖАТИЕ БЕЙСИК-ПРОГРАММ

Существенный недостаток ZX Spectrum — относительно малый объем свободной памяти (около 40 килобайт). А тут еще интерпретатор Бейсика «транжирит» память: под запись любого числа интерпретатор отводит 6 ячеек памяти независимо от размера числа. Даже если это просто единица или ноль. Вот и приходится

* Под каждый символ отводится 8 ячеек памяти.

** Более подробно о подключении альтернативных шрифтов см. [1].

программистам, работающим на Бейсике, изощряться, чтобы втиснуться в узкие рамки памяти Speccy.

Прежде всего надо расширить область памяти, отведенную для работы бейсик-программы. Границы, в которых может размещаться программа с переменными, определяются компьютером, но могут быть изменены. Старший адрес памяти, разрешенный к использованию интерпретатором Бейсика, хранится в системной переменной **RAMTOP** (23730/31). Стандартно он равен 65367. Но если программа не вмещается, его можно увеличить до предела — до 65535*. Для этого нет необходимости записывать новое значение RAMTOP с помощью POKE. Изменить RAMTOP можно с помощью специально предназначенного для этого оператора **CLEAR** с параметром 65535. Правда, CLEAR к тому же очистит все переменные, массивы, выполнит операторы CLS и RESTORE, вообще подготовит программу к новому пуску.

Если и после перемещения RAMTOP не хватает памяти, придется применять более изощренные методы.

Приведем несколько самых распространенных способов сжатия бейсик-программ. Хотя в этой области нет предела для фантазии.

Во-первых, надо свести к минимуму длину имен всех числовых переменных. Если их меньше, чем букв в алфавите, то все сделать однобуквенными.

Памятуя, что числовые константы занимают много места, не помешает самые часто используемые из них заменить на переменные. Надо только единожды в начале программы присвоить им требуемые значения.

Целые числовые константы выгоднее записывать как символьные и помещать их «под» функцию VAL. К примеру, операторы LET a=5 и LET a=VAL "5" тождественны, однако второй оператор занимает в памяти на две ячейки меньше (но, следует учесть, работает медленнее).

Для сокращения объема памяти, занимаемого программой, бывает полезно вместо числовых массивов использовать символьные. Ведь под каждый элемент числового массива, независимо от того, что в нем хранится, отводится по пять ячеек памяти, в то время, как под элемент символьного массива отводится лишь одна ячейка. Преобразовать же символьную константу в число можно с помощью той же VAL.

И, напоследок, парочку «хитростей». Например, вместо нуля, который, как и любое число, занимает в памяти 5 байт, можно записать просто BIN, либо NOT PI. Число «пи» отлично от нуля, и это главное, а логический оператор NOT, по определению, превращает его в ноль. Внешне получилось длиннее и мудреней, но в памяти, тем не менее, такая запись займет лишь две ячейки. Вывернувшись, можно «смоделировать» и единицу. Например, записав NOT BIN либо SNG PI.

---

* При этом, правда, затирается область символов, определяемых пользователем.

ЗАЩИТА БЕЙСИК-ПРОГРАММ

Вообще говоря, коммерческие программы, которые требуют защиты, обычно пишутся не на Бейсике. Но ради интереса немного поговорим и на эту тему.

Простую (не защищенную) программу в любой момент можно запустить, остановить, просмотреть, внести изменения — в общем, сделать с ней все, что душе угодно.

Защита программы может происходить на трех этапах. Во-первых, делают так, чтобы ее невозможно было загрузить в память компьютера без специального загрузчика. Во-вторых, чтобы ее нельзя было нормальным образом остановить (по **BreaK**). И, в-третьих, если уж ее остановили, то нельзя было бы ее просмотреть с помощью LIST.

От просмотра программы можно защититься, расставив в ее тексте коды управления цветом, окрашивающие листинг в одинаковый цвет тона и фона. Например, если после написания программы вызвать на редакцию ее первую строку и нажать комбинацию клавиш **CS/SS+CS/7** (белый цвет тона), то строка «пропадет». После ввода этой строки (**Enter**) бесцветным станет и весь текст программы.

Против принудительного останова программы при нажатии клавиши **Break** (**CS/Space**) можно использовать ненормальную реакцию интерпретатора Бейсика на отсутствие служебного экрана. Надо только в системную переменную DF_SZ записать нулевое значение:

```
POKE 23659,0
```

После нажатия **Break** компьютер должен отреагировать на него выводом сообщения BREAK – CONT **repeats** на служебный экран. Но не тут-то было. Дальше может быть все что угодно, но только не просмотр листинга.

Информации, приведенной в данной главе, явно недостаточно для обсуждения более изощренных способов защиты бейсик-программ и, тем более, способов ее «взламывания». Но мы надеемся еще вернуться к этому разговору в другой книге.

СПРАВОЧНИК ПО SPECTRUM-БЕЙСИКУ

В этом разделе в сжатой и более строгой форме повторено все, что говорилось о Spectrum-Бейсике в первой главе. «Справочник» предназначен для тех, кто постоянно работает с Бейсиком.

Ключевые слова размещаются в алфавитном порядке, для каждого приведены: формат, класс, последовательность набора на клавиатуре, ссылка на страницу основного текста, краткое описание и пример использования. Результаты выполнения примеров, выводимые на экран, выделены более жирным шрифтом. Примеры, набираемые в непосредственном режиме (без номера строки), запускаются нажатием клавиши **Enter**, программы — выполнением оператора RUN. В примерах RUN и **Enter** опущены.

После описания ключевых слов приведены: «Тематический указатель», в котором операторы и функции объединены по сферам их применения, перечень сообщений об ошибках, таблица символов ZX Spectrum и таблица контрольных кодов.

При описании ключевых слов используются обозначения, перечисленные в начале книги, а также:

[E] — расширенный режим клавиатуры (включается одновременным нажатием клавиш **CS** и **SS**).

Классы ключевых слов

По классу ключевые слова подразделяются на операторы, функции и логические операторы:

- **операторы** предписывают компьютеру выполнение определенной последовательности действий, не связанных с преобразованием числовых и символьных значений; формируют программные строки;
- **функции** создают или преобразуют числовые и символьные значения; исходное значение называется *аргументом*, конечное — *результатом*; используются только совместно с операторами;
- **логические операторы** задают действия над условиями.

Терминология

При описании формата операторов и функций применяются следующие понятия:

- **числовая константа** — число в интервале от $2{,}8 \cdot 10^{-39}$ до $1{,}7 \cdot 10^{38}$; точность представления — 8 десятичных знаков;

- **числовая переменная** — имя переменной, хранящей числовую константу, начинается с буквы и может состоять из произвольного количества символов, пробелы игнорируются, заглавные и строчные буквы не различаются;
- **числовое выражение** — допустимая интерпретатором комбинация констант, переменных, функций, объединенных знаками математических операций, в результате вычисления которой получается числовая константа;
- **числовое значение** — числовая константа, числовая переменная, числовое выражение;
- **целочисленное значение** — числовая константа, переменная, выражение, округляющиеся до ближайшего целого;
- **символьная константа** — последовательность символов, заключенная в кавычки (строка символов);
- **символьная переменная** — имя переменной, хранящей символьную константу; составляется из двух символов: буквы и знака $; заглавные и строчные буквы не различаются;
- **символьное выражение** — допустимая комбинация констант, переменных, функций, объединенных математическими операциями, в результате вычисления которой получается символьная константа;
- **символьное значение** — символьная константа, символьная переменная, символьное выражение.

Формат

Формат отражает, какие ключевые слова и значения и в какой последовательности необходимо использовать для составления операторов и функций. При записи формата приняты следующие обозначения:

<z>	–	числовое значение;
<c>	–	целочисленное значение;
<string>	–	символьное значение;
<var>	–	числовая или символьная переменная;
<условие>	–	допустимая комбинация числовых и символьных значений, объединенных знаками сравнения и логическими операторами;
<оператор>	–	допустимый в данном контексте бейсик-оператор;
<addr>	–	значение адреса ячейки памяти (0...65535);
<n>	–	номер строки программы (0...9999)
<row>	–	номер строки экрана (0...21), отсчитывается от верхнего края экрана;
<col>	–	номер столбца экрана (0...31), отсчитывается от левого края экрана;

<x> – горизонтальная координата точки (0...175), отсчитывается от нижнего края основного экрана;
<y> – вертикальная координата точки (0...255), отсчитывается от левого края экрана;
<q> – угол, заданный в радианах;
<color> – код цвета.

Коды цветов

0 – черный	5 – голубой
1 – синий	6 – желтый
2 – красный	7 – белый
3 – фиолетовый	8 – режим прозрачности*
4 – зеленый	9 – режим черно-белого контраста**

Знаки математических операций

+	сложение	=	равно
–	вычитание	<>	не равно
*	умножение	<	меньше
/	деление	>	больше
↑	возведение в степень	<=	меньше или равно
()	скобки	>=	больше или равно

Порядок выполнения операций при расчете выражений

1. выделение подстроки (сечение)
2. функции
3. возведение в степень
4. минус перед числом (отрицательное число)
5. умножение и деление
6. сложение и вычитание
7. сравнение (=, >, <, <=, >=, <>)
8. NOT
9. AND
10. OR

Первыми рассчитываются части выражения, заключенные в скобки.

---

* Не используется в операторе BORDER.
** Только в операторах INK и PAPER.

ABS <z> [E] G

функция стр. 16

модуль числового значения <z>.

Пример: PRINT ABS −3

3

ACS <z> [E] SS/W

функция стр. 15

арккосинус, значение аргумента <z> должно находиться в интервале от −1 до 1.

Пример:* PRINT 180/PI*ACS 0.5

AND SS/Y

логический оператор, функция стр. 35

Логический оператор

реализует **логическое умножение**. Комбинация условий, объединенных AND, истинна, если истинно каждое из условий.

Пример: LET X=10: IF X>5 AND X<15 THEN PRINT "YES"

YES

Функция

- возвращает значение, стоящее перед ключевым словом, если аргумент не равен нулю;
- возвращает ноль или «пустую строку» (в зависимости от того, числовое или символьное значение стоит перед ключевым словом), если аргумент равен нулю.

Пример: PRINT "A" AND 1

A

PRINT 3 AND 0

0

ASN <z> [E] SS/Q

функция стр. 15

арксинус, значение аргумента <z> должно находиться в интервале от −1 до 1.

Пример: PRINT 180/PI*ASN 0.5

---

* Умножением на 180/PI радианы переводятся в градусы.

AT <row>, <col> SS/I

оператор стр. 14

задает знакоместо для вывода данных в операторах PRINT и INPUT. Параметр <row> (0...21) определяет номер строки, <col> (0...31) — номер столбца. Данные отделяются от AT знаками-разделителями (запятая, точка с запятой, апостроф).

Пример: PRINT "AAA"; AT 10,15; "QQQ"

ATN <z> [E] SS/E

функция стр. 15

арктангенс числового значения <z>.

Пример: PRINT 180/PI*ATN 1

ATTR (<row>, <col>) [E] SS/L

функция стр. 55

возвращает значение, соответствующее атрибутам знакоместа. Позиция знакоместа задается двумя целочисленными значениями, записанными через запятую и заключенными в скобки: <row> — номер строки (0...23), <col> — номер столбца (0...31). ATTR возвращает число от 0 до 255, рассчитанное по следующей формуле:

1×<INK>+8×<PAPER>+64×<BRIGHT>+128×<FLASH>,

где <INK>, <PAPER>, <BRIGHT> и <FLASH> — значения атрибутов*.

Пример: Если символ в знакоместе с <row>=11, <col>=16 напечатан синим (код 1) на желтом (6) с повышенной яркостью без мерцания, то функция ATTR (11, 16) вернет число 1+6×8+64+0=112

BEEP <t>, <h> [E] SS/Z

оператор стр. 40

генерирует звук заданной длительности и высоты. Параметр <t> задает длительность звучания в секундах, <h> — высоту в полутонах: 0 соответствует ноте ДО первой октавы, диапазон изменения от −60 до 69. При большой длительности звучания допустимый диапазон высот уменьшается.

Пример: BEEP 0.5, 1: REM ***Нота ДО# первой октавы в течение 0,5 секунды***

---

* По сути, ATTR возвращает байт, в котором три младших разряда (0...2) определяют код цвета тона, следующие три (3...5) — цвета фона, два остальных (6 и 7), соответственно, — BRIGHT и FLASH. Число (0...255), возвращаемое функцией, получается при переводе значения байта из двоичного представления в десятичное: 2^0 × <INK> + 2^3 × <PAPER> + 2^6 × <BRIGHT> + 2^7 × <FLASH>.

BIN <b> [E] B

функция стр. 53

переводит шестнадцатиразрядное двоичное число <b> в десятичное (нули в левой части числа можно опускать).

Пример: PRINT BIN 11111110

254

BORDER <color> B

оператор стр. 22

задает цвет поля вокруг рабочей области экрана и цвет фона служебного экрана. Цвет определяется параметром <color>.

В отличие от операторов INK и PAPER, оператор BORDER не может быть использован в операторе PRINT.

BRIGHT <p> [E] SS/B

оператор стр. 22

устанавливает яркостный режим отображения информации. Параметр <p> может принимать одно из целочисленных значений:

1 – включает режим повышенной яркости отображения информации;

8 – задает режим прозрачности, при котором яркость отображения информации будет повторять яркость, ранее установленную для данного знакоместа, то есть при печати новых символов яркие знакоместа будут оставаться яркими, а нормальные – нормальными;

0 – отменяет режимы повышенной яркости и прозрачности.

Режим яркости устанавливается целиком для знакоместа и одновременно для цвета тона и цвета фона.

В качестве самостоятельного оператора BRIGHT устанавливает тот или иной режим яркости для всех последующих выводов на экран (постоянный атрибут).

BRIGHT может также использоваться в операторах PRINT, INPUT, PLOT, DRAW, CIRCLE, устанавливая режим яркости отображения информации, выводимой только данным оператором (временный атрибут). В операторах PRINT и INPUT ключевое слово BRIGHT размещается в любом месте блока данных, отделяясь от них знаками-разделителями (запятая, точка с запятой, апостроф).

BRIGHT в операторах PLOT, DRAW, CIRCLE ставится сразу за ключевым словом и отделяется от параметров точкой с запятой.

Пример: PRINT "AAA", BRIGHT 1; "AAA"; BRIGHT 0; "AAA"

AAAAAAAAA

CAT

ключевое слово в Spectrum-Бейсике не задействовано.

CHR$ <k> [E] U

функция стр. 18

возвращает символ по указанному в аргументе коду <k> (0...255).

Соответствие символов кодам приведено в табл. 1 на стр. 100.

Коды от 0 до 32 являются управляющими и не имеют символьного представления. Функция CHR$ с этими кодами в качестве аргумента влияет на вывод информации на экран и принтер (см. табл. 2 на стр. 101).

Пример: PRINT CHR$ 65

A

CIRCLE <x>, <y>, <r> [E] SS/H

оператор стр. 20

строит на экране окружность. Параметры окружности задаются в пикселях тремя целочисленными значениями: <x> (0...175) и <y> (0...255) — вертикальная и горизонтальная координаты центра окружности, <r> — радиус.
Цвет окружности соответствует текущим постоянным значениям атрибутов либо задается операторами INK, PAPER и другими, поставленными непосредственно за ключевым словом CIRCLE и отделенными друг от друга и от параметров точкой с запятой.

Пример: CIRCLE INK 3; 128, 88, 87

CLEAR [<addr>] X

оператор стр. 65

обнуляет значения всех переменных, выполняет операторы CLS и RESTORE, сбрасывает позицию PLOT и очищает стек GO SUB.
CLEAR с параметром, кроме того, **заносит значение <addr> в системную переменную RAMTOP** (см. стр. 65).

CLOSE #<c>

оператор используется при работе с потоками и каналами (см. [1]).

CLS V

оператор стр. 21

очищает основной экран и окрашивает его в текущий постоянный цвет фона.

CODE <string> [E] I

функция стр. 18

возвращает код первого символа символьного значения <string> в соответствии с таблицей символов ZX Spectrum (см. табл. 1 на стр. 100). Если аргумент равен «пустой строке», функция возвращает 0.

Пример:
```
PRINT CODE "ABC"

65
```

CONTINUE C

оператор стр. 33

продолжает выполнение программы с оператора, на котором оно было прервано.

COPY Z

оператор стр. 51

распечатывает копию экрана на принтере.

COS <q> [E] W

функция стр. 15

вычисляет косинус угла <q>, заданного в радианах.

DATA <данные>, ... [E] D

оператор стр. 43

задает список данных — произвольный набор числовых и символьных значений. Элементы данных размещаются непосредственно за ключевым словом и отделяются друг от друга запятыми. Оператор DATA можно помещать в любом месте программы. Список данных может быть распределен между несколькими операторами DATA.

Считывание данных (занесение данных в переменные) производится оператором READ. Переменные, используемые в качестве элементов данных, к моменту считывания должны быть определены. Последовательность считывания данных изменяется с помощью оператора RESTORE.

Пример:
```
10 LET D1=31: LET D2=28
20 FOR N=1 TO 2
30 READ X,A$
40 PRINT A$;" ";X;" DAYS"
50 NEXT N
60 DATA D1,"JAN",D2,"FEB"

JAN 31 DAYS FEB 28 DAYS
```

Ключевое слово DATA может быть использовано также в комбинации с LOAD, SAVE, и VERIFY (см. LOAD...DATA, SAVE...DATA и VERIFY...DATA).

DEF FN

[E] SS/1

оператор

определяет пользовательские функции (дополнительно к «встроенным» в Бейсик). Для задания числовой функции после DEF FN ставится буква (имя функции), за ней в скобках одна или несколько отделенных друг от друга запятыми однобуквенных переменных (параметры функции) и следом через знак равенства — числовое выражение, задающее саму функцию.

Пример: 100 DEF FN R(X,Y)=X+Y

Оператор DEF FN помещается в любое место программы.

Вызов пользовательской функции осуществляется ключевым словом FN. Вслед за ним указывается имя функции и в скобках через запятую — значения параметров в порядке, соответствующем их следованию в определении функции.

Пример: PRINT FN R(3,4)

7

В этом примере параметру X присваивается значение 3, а Y — значение 4.

При расчете пользовательской функции переменные, совпадающие по имени с параметрами функции, не изменяются.

В выражении, задающем функцию, могут использоваться и не указанные в скобках переменные. При вызове функции они принимают значения, заданные в программе.

Символьная функция определяется аналогично числовой, только в качестве ее имени используется буква со знаком $.

Пример:

```
150 DEF FN A$(C$,i,j)=C$(i TO j)
200 PRINT FN A$ ("Speccy",2,4)
```

pec

DIM

D

оператор стр. 41

задает массив и его размерность.

Числовые массивы

При задании числового массива вслед за DIM ставится однобуквенное имя массива и в скобках через запятые одно или несколько (в зависимости от размерности массива) целочисленных значений, задающих количество элементов массива по каждому измерению.

Пример:

```
10 DIM Y(5)
20 DIM Z(10,3)
```

В первой строке примера задается одномерный числовой массив, состоящий из 5 элементов (индексированных переменных) Y(1)...Y(5). Во второй — двумерный массив из 30 элементов Z(1,1)...Z(10,3).

После задания массива его элементам присваивается нулевое значение.

Число измерений массива не может превышать 255.

Параллельно с числовыми массивами в программе могут использоваться неиндексированные числовые переменные с тем же именем.

Символьные массивы

Имена символьных массивов состоят из буквы и знака $. При определении массива в скобках после имени, кроме числовых значений, задающих его размерность, указывается число, определяющее длину элементов массива (количество символов в них).

Пример:
```
10 DIM Z$(20,10)
20 DIM C$(10,4,5)
```

В строке 10 задается одномерный символьный массив из 20 элементов Z$(1)...Z$(20), каждый из которых представляет собой строку длиной в 10 символов. В строке 20 — двумерный массив из 40 строк длиной по 5 символов.

N-мерный символьный массив можно рассматривать как (N+1)-мерный массив с элементами длиной в один символ. Следовательно, можно выделить любой символ элемента массива, указав в индексированной переменной, кроме значений индексов, еще и номер позиции символа в строке.

Пример:
```
70 LET Z$(2)="QWERTYUIOP"
75 PRINT Z$(2,3)

E
```

После определения символьного массива его элементам присваивается значение строки, состоящей из пробелов. Символьные значения, присваиваемые элементам массива, либо дополняются пробелами до нужной длины, либо усекаются.

В отличие от числовых массивов, одновременно с символьным массивом не может существовать неиндексированная символьная переменная с тем же именем.

Символьные массивы нулевой размерности

Можно определить символьный массив нулевой размерности, указав в скобках после имени массива только одно значение.

Пример:
```
10 DIM A$(15)
```

Заданный массив имеет один элемент A$ — строку фиксированной длины, равной указанному в скобках значению, в данном случае — 15 символов.

DRAW <a>, <b>[, <q>] W

оператор стр. 20

строит на экране отрезок прямой линии или дугу окружности.

Для проведения прямой линии следом за DRAW через запятую ставятся два целочисленных значения, задающих относительное расстояние в пикселях от начала отрезка до его конца, соответственно, по горизонтали (<a>) и вертикали (<b>). При проведении отрезка влево и вниз эти значения должны быть отрицательными. За начало отрезка принимается точка, выставленная последним выполненным графическим оператором (DRAW, PLOT, CIRCLE). После включения компьютера начальная точка построения линий устанавливается в левый нижний угол основного экрана (0, 0).

Для проведения дуги окружности после значений, определяющих расстояние от начальной точки построения до конечной, через запятую указывается числовое значение <q>. Оно задает угол в радианах, получающийся при соединении концов дуги с центром окружности, частью которой она является. При положительном значении <q> дуга строится против часовой стрелки, при отрицательном — по часовой.

Цвета линии соответствуют текущим постоянным значениям атрибутов либо задаются операторами INK, PAPER и другими, поставленными непосредственно за ключевым словом DRAW и отделенными друг от друга и от параметров точками с запятой.

Пример: DRAW INK 2; 80, 50, –0.5

ERASE

ключевое слово в Spectrum-Бейсике не задействовано.

EXP <z> [E] X

функция стр. 15

экспонента: возводит число e=2,7182818 в степень, заданную числовым значением <z>.
Пример: PRINT EXP 1

2.7182818

FLASH <p> [E] SS/V

оператор стр. 22

устанавливает режим мерцания при отображении символьной и графической информации. Параметр <p> может принимать одно из целочисленных значений:

1 – включает режим мерцания. Изображение на экране компьютера при включенном режиме периодически меняет цвет тона на цвет фона и обратно;

8 – задает режим прозрачности, при котором режим мерцания будет повторять режим, ранее установленный для данного знакоместа, то есть при печати новых символов мерцающие знакоместа будут оставаться мерцающими, а нормальные — нормальными;

0 – выключает мерцание и режим прозрачности.

FLASH устанавливает режим мерцания целиком для знакоместа.

В качестве самостоятельного оператора FLASH задает режим мерцания для всех последующих выводов на экран (постоянный атрибут).

FLASH может также использоваться в операторах PRINT, INPUT, PLOT, DRAW, CIRCLE, устанавливая режим мерцания графического и символьного изображения, выводимого только данным оператором (временный атрибут).

В операторах PRINT и INPUT ключевое слово FLASH размещается в любом месте перед данными, отделяясь от них знаками-разделителями (запятая, точка с запятой, апостроф). FLASH в операторах PLOT, DRAW, CIRCLE ставится сразу за ключевым словом и отделяется от параметров точкой с запятой.

Пример: PRINT "A"; FLASH 1; "B"; FLASH 0; "C"

FN [E] SS/2

функция

возвращает значение пользовательской функции, заданной оператором DEF FN. См. описание DEF FN.

FOR <i>=<a> TO <b> [STEP <s>] F

оператор стр. 37

определяет начало и параметры цикла FOR...NEXT, позволяющего выполнять часть программы, заключенную между операторами FOR и NEXT, заданное число раз. Вслед за FOR ставится буква — *управляющая переменная цикла* (<i>), знак равенства и два числовых значения, разделенных ключевым словом TO: <a> задает начальное значение переменной цикла, <b> — конечное. Далее может следовать ключевое слово STEP со стоящим за ним числовым значением <s>, задающим *шаг* приращения управляющей переменной. Шаг может быть как целым, так и дробным, как положительным, так и отрицательным. При отсутствии ключевого слова STEP шаг устанавливается равным 1.

Пример: 50 FOR i=1 TO 25 STEP 5

После конструкции FOR...TO...STEP размещается *тело цикла* — фрагмент программы, который требуется повторить заданное число раз. В теле цикла может использоваться переменная цикла. Тело цикла завершается оператором NEXT со стоящей за ним переменной цикла.

Пример: 100 NEXT i

Оператор NEXT увеличивает значение переменной цикла на заданный шаг и сравнивает ее новое значение с конечным значением <b>. Если переменная цикла не превысила конечное значение, то тело цикла выполняется еще раз, в противном случае цикл завершается и программа переходит к выполнению следующего за NEXT оператора.

Возможно построение неограниченного числа вложенных циклов FOR...NEXT.

FORMAT

ключевое слово в Spectrum-Бейсике не задействовано.

GO SUB <n> H

оператор стр. 45

передает управление подпрограмме. Номер строки, с которой начинается подпрограмма, задается целочисленным значением <n>. Текст подпрограммы должен заканчиваться оператором RETURN, который возвращает управление оператору, следующему за GO SUB.

Из подпрограммы может осуществляться вызов другой подпрограммы. Количество таких вызовов (вложений) не ограничено.

Пример:
```
25 GO SUB 1000
30 GO SUB 1000+X
```

GO TO <n> G

оператор стр. 32

передает управление строке программы. Номер строки задается целочисленным значением <n>. В режиме непосредственного исполнения GO TO используется для запуска программы с любой строки без обнуления переменных и очистки экрана.

IF <условие> THEN <оператор>[:<оператор>] ... U

оператор стр. 33

передает управление последовательности операторов при выполнении заданного условия. Условие ставится за IF, последовательность операторов — после ключевого слова THEN. При невыполнении (ложности) условия управление сразу передается на следующую строку программы.

Условием может быть и числовое значение. При этом считается, что истинным значением является отличное от нуля значение, а ложным — равное нулю.

Несколько условий может быть объединено логическими операторами в логическое выражение.

Пример:
```
50 IF X<3 THEN PRINT "KU-KU": GO TO 250
75 IF X+Y/5 THEN PRINT X+5;
85 IF A$="AAA" AND A=25 THEN GO SUB 100
```

IN <addr> [E] SS/I

функция

считывает число (байт) из порта (см. [1]). Адрес порта задается целочисленным значением <addr> (0...65535), записываемым за ключевым словом.

Пример: 150 LET X=IN Y

INK <color> [E] SS/X

оператор стр. 21

устанавливает цвет тона — цвет, в который окрашиваются выводимые на экран символы, точки, линии. Вслед за INK указывается целочисленное значение <color> в интервале от 0 до 9, которым кодируется цвет.

После выполнения оператора INK 8 выводимые на экран символы и графические объекты будут окрашиваться в цвета, ранее установленные для данных знакомест (режим прозрачности). INK 9 устанавливает белый или черный цвет тона в зависимости от цвета фона с целью достижения наибольшего контраста (режим черно-белого контраста).

В качестве самостоятельного оператора INK устанавливает постоянный цвет тона, влияющий на все последующие выводы на экран (постоянный атрибут).

INK может также использоваться в операторах PRINT, INPUT, PLOT, DRAW и CIRCLE, устанавливая временный цвет тона. В этом случае INK влияет на цвет символов и графических объектов, выводимых только данным оператором (временный атрибут). В операторах PRINT и INPUT ключевое слово INK размещается в любом месте блока данных, отделяясь от них знаками-разделителями (запятая, точка с запятой, апостроф).

Пример: PRINT "DU-DU-DU";INK 3;"KU-KU"

INK в операторах PLOT, DRAW, CIRCLE ставится сразу за ключевым словом и отделяется от параметров точкой с запятой.

Пример: CIRCLE INK 4; 128,88,87

INKEY$ [#<c>] [E] N

функция стр. 39

возвращает символ, соответствующий клавише, нажатой в момент выполнения функции (прописные и строчные буквы различаются), либо «пустую строку», если ни одна из клавиш не нажата. Аргумента не требует. Используется для обнаружения факта нажатия клавиши на клавиатуре.

С параметром #<c> функция используется при работе с потоками и каналами (см. [1]).

Пример: 60 IF INKEY$<>"q" THEN GO TO 60: *REM Останавливает работу программы до тех пор, пока не будет нажата клавиша* **q** *(без* **CS***)*

INPUT

I

оператор стр. 31

вводит данные в программу: присваивает числовым или символьным переменным, расположенным за ключевым словом, соответственно, числовые или символьные значения, набираемые на клавиатуре. Вводимые значения могут содержать определенные в программе переменные. Следом за INPUT в произвольном порядке размещаются переменные любого типа и текст комментариев. Текст комментариев составляется из символьных и числовых констант или переменных. Комментарии, содержащие переменные, заключаются в скобки.

Элементы, следующие за оператором INPUT, отделяются друг от друга знаками-разделителями (запятая, точка с запятой, апостроф), действие которых аналогично их действию в операторе PRINT. В INPUT также могут быть использованы ключевые слова INK, PAPER, FLASH, BRIGHT, AT и TAB.

Пример: 10 INPUT INK 2;"WHAT IS YOUR NAME? ";N$, ("HOW OLD ARE YOU,"+N$+"? ");AGE

Комментарии и вводимые данные оператор INPUT отображает в нижней строке служебного экрана. После заполнения строки текст смещается на строку выше.

Встретив переменную, не входящую в текст комментария (расположенную вне скобок), компьютер выводит на экран курсор и ожидает ввода данных. При ожидании символьных данных на экран выводятся две кавычки. Данные набираются на клавиатуре и отображаются на экране. Они могут редактироваться подобно строкам бейсик-программы. После нажатия клавиши **Enter** набранное значение присваивается переменной.

Введя оператор STOP на запрос ввода числовой константы, можно остановить выполнение программы. Останов программы в момент ожидания символьных данных также производится вводом STOP, но предварительно нужно удалить первую кавычку.

Избежать появления кавычек при ожидании ввода символьных данных можно, используя конструкцию INPUT LINE. Ключевое слово LINE ставится перед символьной переменной. Комментарий помещается между INPUT и LINE. Прерывание работы оператора INPUT...LINE производится нажатием клавиш **CS/6**.

Пример: 70 INPUT "WHAT IS YOUR NAME? "; LINE N$

Используя параметр #<c> в произвольном месте списка параметров и данных оператора INPUT, можно переключать ввод и вывод информации на другие потоки (см. [1]).

INT <z> [E] R

функция стр. 16

округляет числовое значение <z> до ближайшего меньшего целого числа.

Пример: PRINT INT 45.67;" ";INT −7.66

45 −8

INVERSE <p> [E] SS/M

оператор стр. 22

устанавливает режим инвертированного отображения графической и символьной информации. Следом за INVERSE ставится целочисленное значение <p>, которое может равняться либо 0, либо 1.

После выполнения оператора INVERSE 1 при печати символов цвет тона и цвет фона меняются местами. Графическая информация при включенном режиме инверсии отображается цветом фона и становится невидимой. Оператор INVERSE 0 устанавливает нормальный режим цветопередачи.

В качестве самостоятельного оператора INVERSE устанавливает режим инвертирования для всех последующих выводов на экран.

INVERSE может также использоваться в операторах PRINT, INPUT, PLOT, DRAW, CIRCLE. При этом устанавливается режим инвертирования графической и символьной информации, выводимой только данным оператором. В операторах PRINT и INPUT ключевое слово INVERSE размещается в любом месте перед данными, отделяясь от них знаками-разделителями (запятая, точка с запятой, апостроф).

INVERSE в операторах PLOT, DRAW, CIRCLE ставится сразу за ключевым словом и отделяется от параметров точкой с запятой.

LEN <string> [E] K

функция стр. 17

возвращает длину (число символов) символьного значения.

Пример: 120 INPUT A$; IF LEN A$>9 THEN GO TO 120: REM *«Пропускаются» строки длиной только до 9 символов*

LET <var>= L

оператор стр. 28

присваивает значение переменной. После LET ставится переменная, следом за которой через знак равенства записывается присваиваемое ей значение.

Пример:
```
60 LET X=X+1
80 LET A$="PITER"+STR$ X
90 LET S(5,10)=LEN A$
```

До присвоения переменной значения (операторами LET, READ или INPUT) она является неопределенной и не может быть использована в программе.

LINE [E] SS/3

См. SAVE, INPUT.

LIST [<n>] K

оператор стр. 27

отображает на экране листинг бейсик-программы, находящейся в памяти компьютера. LIST без каких-либо значений после ключевого слова начинает вывод листинга с начала программы. Указав после LIST номер строки <n>, можно начать вывод листинга с требуемого места программы. При этом указатель текущей строки устанавливается на строку с номером <n>.

После заполнения экрана вывод листинга приостанавливается и может быть продолжен после нажатия любой клавиши, кроме **N**, **Space**, **Break**. Перечисленные клавиши прерывают выдачу листинга.

LIST #<c>

оператор используется при работе с потоками и каналами (см. [1]).

LLIST [<n>] [E] V

оператор стр. 51

распечатывает на принтере листинг бейсик-программы, находящейся в памяти компьютера, начиная со строки <n>. LLIST без параметров распечатывает листинг с начала программы.

LLIST #<c>

оператор используется при работе с потоками и каналами (см. [1]).

LN <z> [E] Z

функция стр. 15

натуральный логарифм положительного числового значения <z>.

LOAD <string> J

оператор стр. 47

загружает в память компьютера бейсик-программу с магнитной ленты. Имя загружаемой программы задается символьным значением <string>. LOAD с «пустым» именем загружает первую встретившуюся на ленте программу. При загрузке ранее находящаяся в памяти программа уничтожается.

Пример:
```
LOAD "FILENAME"
LOAD N$
LOAD ""
```

LOAD <string> **CODE** [<addr>][,<c>] J...[E] I

оператор стр. 50

загружает с магнитной ленты в память компьютера блок кодов (последовательность байтов). Имя файла, в котором был сохранен блок на магнитной ленте, задается символьным значением <string>.

За ключевым словом CODE указываются через запятую два целочисленных значения: <addr> — адрес размещения блока кодов в памяти компьютера и <c> — количество загружаемых байт. При отсутствии параметра <c> по приведенному адресу загружается весь блок.

Без указания числовых значений оператор LOAD...CODE загружает блок кодов по адресу, с которого он был записан на магнитную ленту оператором SAVE...CODE.

Пример:
```
LOAD "NAME" CODE
LOAD "PICTURE" CODE 16384, 6912
LOAD "" CODE
```

LOAD <string> **DATA** <a>[$]() J...[E] D

оператор стр. 49

загружает с магнитной ленты в память компьютера массив. Имя файла, содержащего массив, задается символьным значением <string>. Следом за ключевым словом DATA указывается однобуквенное имя массива <a> с пустыми скобками. Значения, считываемые с ленты, загружаются в индексированные переменные этого массива.

Пример:
```
LOAD "NUMBER" DATA N()
LOAD "NAMES" DATA N$()
```

LOAD <string> **SCREEN$** J...[E] SS/K

оператор стр. 49

загружает с магнитной ленты блок кодов непосредственно в экранную область памяти, при этом на экране воспроизводится ранее записанная картинка. Имя файла, в котором она была сохранена на ленте, указывается символьным значением <string>.

Пример:
```
LOAD "PICTURE" SCREEN$
```

LPRINT [E] C

оператор стр. 51

распечатывает данные на печатающем устройстве.

Формат оператора зависит от типа принтера и программы, его обслуживающей (драйвера принтера).

Пример:
```
LPRINT "ABCDEF"
```

LPRINT #<c>

оператор используется при работе с потоками и каналами (см. [1]).

MERGE <string> [E] SS/T

оператор стр. 48

подгружает бейсик-программу с магнитной ленты в память компьютера, не стирая уже находящейся там программы. Имя загружаемой программы задается символьным значением <string>. Оператор MERGE с двумя пустыми кавычками подгружает первую встретившуюся на ленте программу.

Оператор MERGE соединяет две программы, располагая их строки по порядку возрастания номеров. При совпадении номеров строк «старые» строки заменяются на «новые».

MOVE

ключевое слово в Spectrum-Бейсике не задействовано.

NEW A

оператор

очищает область памяти, отведенную для размещения и работы бейсик-программы (область памяти до адреса, занесенного в системную переменную RAMTOP). Параметров не требует.

NEXT <i> N

оператор стр. 37

ограничивает тело цикла FOR...NEXT (не используется отдельно от ключевого слова FOR). См. FOR.

NOT SS/S

логический оператор, функция стр. 35

Логический оператор

реализует **логическое отрицание условия**, стоящего за ключевым словом: истинное условие делает ложным, ложное — истинным.

Пример: 90 LET X=2: IF NOT X>10 THEN PRINT "YES"

YES

Функция

возвращает 0, если числовое значение, следующее за ключевым словом, не равно нулю; 1 — если значение равно нулю.

Пример: PRINT NOT(5*5—25)

1

OPEN #<c>, "N"

оператор используется при работе с потоками и каналами (см. [1]).

OR SS/U

логический оператор, функция стр. 35

Логический оператор

логическое сложение условий. Комбинация условий, объединенных одним или несколькими операторами OR, истинна, если истинно хотя бы одно из условий.

Пример: LET X=20: IF X=50 OR X=20 THEN PRINT "YES"

YES

Функция

- возвращает значение, стоящее перед ключевым словом, если ее аргумент равен 0;
- возвращает 1, если аргумент не равен 0.

Пример: PRINT 45 OR 5

1

OUT <addr>, <byte> [E] SS/0

оператор

записывает в порт целочисленное значение <byte> (0...255), адрес которого задается целочисленным значением <addr> (0...65535).

Пример: OUT 254,3

OVER <p> [E] SS/N

оператор стр. 22

устанавливает режим наложения изображений на экране. Целочисленное значение <p> может принимать значения:

1 – изображение (символьное или графическое) накладывается на изображение, уже существующее на экране, не стирая его; точки пересечения изображений принимают окраску фона;

8 – задает режим прозрачности, при котором режим наложения соответствует ранее установленному для данного знакоместа;

0 – устанавливает нормальный режим отображения информации.

В качестве самостоятельного оператора OVER устанавливает режим наложения для всех последующих выводов на экран.

Ключевое слово OVER может также размещаться в операторах PRINT, INPUT, PLOT, DRAW, CIRCLE. При этом режим наложения задается для графической и символьной информации, выводимой только данным оператором. В операторах PRINT и INPUT ключевое слово OVER

размещается в любом месте блока данных, отделяясь от них знаками-разделителями (запятая, точка с запятой, апостроф).

OVER в операторах PLOT, DRAW, CIRCLE ставится за ключевым словом и отделяется от параметров точкой с запятой.

Пример: PRINT AT 11,15; "YES"; OVER 1; AT 11,15; "____"

YES

PAPER <color> [E] SS/C

оператор стр. 21

устанавливает цвет фона — цвет, в который окрашивается экран. Следом за PAPER указывается целочисленное значение <color> в интервале от 0 до 9, которым кодируется цвет.

После выполнения оператора PAPER 8 при печати в знакоместо цвет фона будет принимать значение, ранее установленное для данного знакоместа. PAPER 9 устанавливает белый или черный цвет фона в зависимости от цвета тона с целью достижения наибольшего контраста.

В качестве самостоятельного оператора PAPER устанавливает постоянный цвет фона, влияющий на все последующие выводы на экран (постоянный атрибут).

PAPER может также использоваться в операторах PRINT, INPUT, PLOT, DRAW, CIRCLE, устанавливая временный цвет фона. В этом случае PAPER влияет на фон знакомест, в которые выводится информация только данным оператором (временный атрибут). В операторах PRINT и INPUT ключевое слово PAPER размещается в любом месте перед данными, отделяясь от них знаками-разделителями (запятая, точка с запятой, апостроф).

Пример: PRINT INK 1;"DI-DI-DI";PAPER 5;"KU"

PAPER в операторах PLOT, DRAW, CIRCLE ставится сразу за ключевым словом и отделяется от параметров точкой с запятой.

PAUSE <t> M

оператор стр. 39

приостанавливает работу программы на время, задаваемое целочисленным значением <t> (0...65535). Пауза исчисляется в 1/50 сек. Прервать паузу можно нажатием любой клавиши. Оператор PAUSE 0 продолжит работу программы только после нажатия клавиши.

PEEK <addr> [E] 0

функция стр. 60

считывает число в интервале от 0 до 255 (байт) из ячейки памяти. Адрес ячейки задается целочисленным значением <addr> (0...65535).

PI [E] M

функция стр. 16

возвращает значение числа «пи» (3,1415927). Аргументов не требует.

PLOT <x>, <y> Q

оператор стр. 19

ставит на экран точку (включает пиксель). Позиция точки задается двумя целочисленными значениями, следующими через запятую после ключевого слова: <x> (0...255) определяет горизонтальную координату, <y> (0...175) — вертикальную.

Цвета точки и фона знакоместа, в которое она помещается, определяются постоянными значениями атрибутов либо задаются временными атрибутами. Ключевые слова, задающие временные атрибуты, размещаются непосредственно за ключевым словом PLOT и отделяются друг от друга и от параметров точками с запятой.

Пример: 160 PLOT INK 2; INVERSE 1; x,y

POINT (<x>, <y>) [E] SS/8

функция

проверяет состояние пикселя. Если пиксель включен (окрашен в цвет тона), функция возвращает 1 и если выключен (окрашен в цвет фона), — 0. Координаты пикселя задаются двумя целочисленными значениями, следующими после ключевого слова в скобках через запятую: <x> (0...255) — горизонтальная координата, <y> (0...175) — вертикальная.

Пример: 240 IF POINT (X,Y)=1 THEN GO SUB 600

POKE <addr>, <byte> O

оператор стр. 62

записывает в память целочисленное значение <byte> (0...255). Адрес ячейки памяти задается целочисленным значением <addr> (0...65535).

Пример: POKE 23609,255

PRINT P

оператор стр. 8

выводит на экран данные — числовые и символьные значения, следующие за ключевым словом. Перед блоками данных в любом порядке могут ставиться ключевые слова, задающие позицию, и атрибуты выводимых на экран символов (TAB, AT, INK, PAPER, FLASH, BRIGHT, INVERSE и OVER). Друг от друга ключевые слова и блоки данных отделяются знаками-разделителями: точкой с запятой, запятой или апострофом.

Символьные значения оператор PRINT освобождает от кавычек и выводит на экран. Числовые выражения, стоящие за PRINT, вычисляются, и на экране отображается результат в десятичной форме (не более восьми значащих цифр). Числа больше 10^8 и меньше 10^{-5} отображаются в экспоненциальной форме.

Пример: PRINT 500000000

5E8

PRINT .0000092

9.2E–6

Блоки данных, разделенные точкой с запятой, отображаются оператором PRINT один за другим без пробелов. Запятая переводит текущую позицию печати на середину текущей строки или в начало следующей, в зависимости от позиции последнего напечатанного символа. Данные, отделенные апострофом, отображаются с новой строки.

Знак, стоящий в конце оператора PRINT, оказывает действие на последующий оператор PRINT.

Пример: PRINT 10–9;2,3'"ASD"

```
12          3
ASD
```

Используя параметр #<c> в произвольном месте списка параметров и данных оператора PRINT, можно переключать вывод информации на другие потоки (см. [1]).

RANDOMIZE [<c>] T

оператор стр. 54

задает начало последовательности случайных чисел, генерируемой оператором RND. Целочисленное значение <c> (1...65535) задает фиксированное начало последовательности. RANDOMIZE без параметра либо с нулевым значением задает начало последовательности в зависимости от содержимого системной переменной FRAMES (см. стр. 61). При этом достигается относительная случайность начала последовательности.

READ <var>[, <var>]... [E] A

оператор стр. 43

присваивает переменным значения, считываемые из списка данных, следующих за операторами DATA. Переменные указываются через запятую за ключевым словом. Тип переменных (числовые или символьные) должен соответствовать типу DATA-данных.

Первый в программе оператор READ последовательно считывает данные, начиная с первого элемента первого встреченного в программе оператора DATA, и присваивает их расположенным за ним переменным. Следующий READ продолжает считывание данных.

Последовательность считывания данных может быть изменена оператором RESTORE.

Пример: 20 READ A$, X

REM E

оператор стр. 31

отделяет текст комментариев от текста программы. Следом за REM могут стоять любые символы.

Пример: 50 GO TO 1000: REM *Переход на конец программы*

RESTORE [<n>] [E] S

оператор стр. 44

указывает оператор DATA, с которого нужно считывать данные следующим по программе оператором READ. Строка, в которой расположен требуемый оператор DATA, задается целочисленным значением <n>. Если следом за RESTORE не указано никакого значения либо оно равно 0, следующий оператор READ обращается к первому в программе оператору DATA.

RETURN Y

оператор стр. 45

передает управление оператору, следующему за последним выполненным оператором GO SUB.

RND [E] T

функция стр. 54

возвращает псевдослучайное число в интервале: больше или равно 0 и меньше 1. Аргументов не требует.

После включения (сброса) компьютера или выполнения оператора NEW функция RND генерирует одну и ту же последовательность случайных чисел. Изменить последовательность можно выполнением оператора RANDOMIZE.

Пример: FOR I=1 TO 22: PRINT RND: NEXT I

RUN [<n>] R

оператор стр. 26

запускает выполнение бейсик-программы. Строка, с которой нужно начать выполнение, задается целочисленным значением <n>. RUN без параметра запускает программу со строки с наименьшим номером.

Перед запуском программы оператор RUN выполняет CLEAR, то есть обнуляет все переменные и т. д. (см. CLEAR).

SAVE <string> [**LINE** <n>] S

оператор стр. 46

записывает бейсик-программу на магнитную ленту. Программе присваивается имя, задаваемое символьным значением <string>. Имя должно состоять не более чем из десяти символов и не может быть «пустым». При выполнении оператора компьютер выдает сообщение: **Start tape, then press any key** (включите магнитофон, затем нажмите любую клавишу).

В операторе SAVE ключевое слово LINE, стоящее после имени программы, указывает, что программа после ее загрузки оператором LOAD будет сразу запущена со строки с номером <n> (автоматически выполнится оператор GO TO <n>).

Пример: SAVE "NAME" LINE 1

SAVE <string> **CODE** <addr>, <c> S...[E] I

оператор стр. 50

сохраняет на магнитной ленте блок кодов (последовательность байтов). Имя файла, в который записывается блок, задается символьным значением <string>. Имя должно состоять не более чем из десяти символов и не может быть «пустым».

За ключевым словом CODE указываются через запятую два целочисленных значения: <addr> (0...65535) — адрес размещения блока кодов в памяти компьютера и <c> — количество входящих в него байт.

Пример: SAVE "PICTURE" CODE 16384,6912

SAVE <string> **DATA** <a>[$]() S...[E] D

оператор стр. 49

сохраняет на магнитной ленте массив. Имя файла, в котором сохраняется массив, задается символьным значением <string>. Имя должно состоять не более чем из десяти символов и не может быть «пустым». Затем следуют ключевое слово DATA и имя массива <a> (буква или буква со знаком $) с пустыми скобками.

Пример: 450 SAVE "NUMBERS" DATA N()

SAVE <string> **SCREEN$** S...[E] SS/K

оператор стр. 49

записывает на магнитную ленту экранное изображение. Имя файла, в котором сохраняется экранное изображение, задается символьным значением <string>. Имя должно состоять не более чем из десяти символов и не может быть «пустым».

SCREEN$ (<row>, <col>) [E] SS/K

функция

возвращает символ, помещенный в указанном знакоместе экрана. Позиция знакоместа задается двумя целочисленными значениями, следующими в скобках через запятую следом за ключевым словом: <row> (0...21) — номер строки, <col> (0...31) — номер столбца.

Возвращаются только символы набора, на который указывает системная переменная CHARS. При несовпадении изображения в знакоместе ни с одним из символов набора функция SCREEN$ возвращает «пустую строку».

Пример: 160 IF SCREEN$(14,19)="#" THEN PRINT AT 14,19; "$"

SGN <z> [E] F

функция стр. 16

определяет знак числового значения. SNG возвращает:

 1 – если аргумент положителен,
–1 – если аргумент отрицателен,
 0 – если аргумент равен 0.

SIN <q> [E] Q

функция стр. 15

вычисляет синус угла <q>, заданного в радианах.

SQR <z> [E] H

функция стр. 15

вычисляет квадратный корень числового значения <z>.

STEP SS/D

оператор стр. 37

используется для построения цикла FOR...NEXT (см. FOR).

STOP SS/A

оператор стр. 33

останавливает выполнение бейсик-программы. Параметров не требует. После остановки программы ее выполнение может быть продолжено оператором CONTINUE с оператора, следующего за STOP.

STR$ <z> [E] Y

функция стр. 17

преобразует числовое значение <z> в символьное. STR$ возвращает значение аргумента в виде символьной константы.

Пример: PRINT LEN STR$ 100

```
3
```

TAB <col> [E] P

оператор стр. 14

задает позицию вывода на экран в текущей строке. Целочисленное значение <col> (0...31) указывает номер колонки. Если <col> меньше текущей позиции вывода, символ помещается в заданную позицию, но строкой ниже. TAB используется только совместно с операторами PRINT и INPUT.

Пример: PRINT TAB 1;"A";TAB 5;"B";TAB 3;"C"

```
 A   B
   C
```

TAN <q> [E] E

функция стр. 15

вычисляет тангенс угла <q>, заданного в радианах.

THEN SS/G

См. IF.

TO SS/F

оператор, функция стр. 16

Оператор

используется для построения циклов FOR...NEXT (см. FOR).

Функция

выделяет подстроку (выполняет сечение) символьного значения. Начальная и конечная позиции сечения задаются двумя целочисленными значениями, которые записываются через ключевое слово TO в скобках следом за символьным значением.

По умолчанию первый аргумент принимает значение, равное 1, второй — равное длине символьного значения.

Пример: PRINT "DU-DU"(2 TO 4)

```
U-D
```

USR [E] L

функция стр. 56

С целочисленным значением <addr>

передает управление подпрограмме в машинных кодах, расположенной по адресу <addr> (0...65535). После выполнения подпрограммы возвращает содержимое регистровой пары процессора BC.

Пример: RANDOMIZE USR 65000

С символьным значением

возвращает адрес первого из восьми байтов, задающих изображение символа, определяемого пользователем. Следом за USR указывается односимвольное значение, соответствующее одной из 21 клавиш (от A до U).

Пример: PRINT USR "a"

65368

VAL <string> [E] J

функция стр. 17

преобразует символьное значение, представляющее собой символьную запись числового значения, в числовую константу.

Пример: PRINT VAL "3+5"−5

3

VAL$ "<string>" [E] SS/J

функция

возвращает строку символов, представляющую собой результат вычисления символьного выражения, записанного в виде символьной константы.

Пример:

```
10 LET A$="ss": LET B$="rr"
20 LET C$="A$+""D""+B$"
20 PRINT C$, VAL$ C$
```

A$+"D"+B$ ssDrr

VERIFY <string> [E] SS/R

оператор стр. 47

проверяет правильность записи программы на магнитную ленту. Имя проверяемой программы задается символьным значением <string>. Если одноименные программы: программа, записанная на магнитной ленте, и программа, находящаяся в памяти компьютера, не совпадают, то выдается сообщение об ошибке: Tape loading error.

С помощью операторов VERIFY...DATA, VERIFY...SCREEN$ и VERIFY...CODE можно проверить правильность записи массивов, экранного изображения и блоков кодов, указав после имени программы соответствующее ключевое слово с параметрами.

Тематический указатель

Атрибуты экрана

Алгебраические функции

Пользовательские функции

Логические операции

Случайные числа

Символьные (строковые) функции

Операции с ячейками памяти

Работа с портами

Работа с магнитофоном

Загрузка программ

Сохранение программ

Вывод на печатающее устройство

Сообщения об ошибках

При обнаружении ошибки интерпретатор останавливает выполнение программы (или бейсик-оператора, набранного в непосредственном режиме) и выводит в нижней части экрана соответствующее сообщение. Каждое сообщение содержит код (цифру или букву), указание на причину ошибки и номер строки и оператора в ней, на котором остановилось выполнение программы.

0 OK

Успешное завершение программы или попытка перейти на строку с номером, большим любого, имеющегося в программе.

1 NEXT without FOR — NEXT без FOR

встречен оператор NEXT без соответствующего FOR.

2 Variable not found — переменная не найдена

переменная была использована без присвоения ей значения или загрузки значения с ленты.

3 Subscript wrong — неправильный индекс

значение индекса вышло за границу размерности массива.

4 Out of memory — нет памяти

не хватает памяти для выполнения оператора.

5 Out of screen — выход за экран

оператор INPUT сгенерировал более 22 строк в нижней части экрана, или в AT был использован номер строки, превышающий 21.

6 Number to big — число слишком велико

в результате вычислений получается число больше $1{,}7 \cdot 10^{38}$ (возможно, была попытка деления на 0).

7 RETURN without GOSUB — RETURN без GO SUB

число операторов RETURN больше числа GO SUB.

8 End of file — конец файла

сообщение в Spectrum-Бейсике не используется.

9 STOP statement — оператор STOP

был использован оператор STOP для остановки программы. CONTINUE продолжит выполнение со следующего оператора.

A Invalid argument — недопустимый аргумент

функция получила недопустимое значение аргумента.

B Integer out of range — округление вышло за границы

значение было округлено до ближайшего целого и не попало в допустимый диапазон.

C Nonsense in BASIC — бессмысленно в Бейсике

оператор не имеет смысла с точки зрения Spectrum-Бейсика.

D BREAK – CONT repeats — прерывание – CONTINUE повторит

останов по клавише **Break**. CONTINUE повторит оператор, при выполнении которого произошло прерывание.

E Out of DATA — нет данных

количество считываний данных операторами READ превысило количество элементов данных в операторах DATA.

F Invalid file name — недопустимое имя файла

в операторе SAVE указано недопустимое имя файла (меньше одного символа или больше 10 символов) либо использовано недопустимое имя канала в операторе OPEN# (см. [1]).

G No room for line — нет места для строки

не хватает свободной памяти для ввода новой строки программы.

H STOP in INPUT — STOP в операторе INPUT

ввод STOP на запрос оператора INPUT.

I FOR without NEXT — FOR без NEXT

число операторов NEXT меньше числа операторов FOR, и одновременно некорректно заданы предельное значение управляющей переменной и шаг ее изменения.

J Invalid I/O device — недопустимое устройство ввода/вывода

сообщение, используемое при работе с каналами и потоками (см. [1]).

K Invalid colour — недопустимый цвет

задано недопустимое значение в INK, PAPER, FLASH, INVERSE или OVER, или использован недопустимый управляющий символ.

L BREAK into program — прерывание программы

нажата клавиша **Break**. В сообщении указывается оператор, выполненный последним. Оператор CONTINUE продолжит выполнение программы со следующего оператора.

M RAMTOP no good — недопустимый RAMTOP

значение, занесенное в системную переменную RAMTOP, слишком мало.

N Statement lost — нет оператора

была предпринята попытка перехода на несуществующий оператор.

O Invalid stream — недопустимый поток

сообщение, используемое при работе с потоками (см. [1]).

P FN without DEF — FN без DEF

вызов функции пользователя без ее определения.

Q Parametr error — ошибка в параметрах

оператор FN содержит неверное количество параметров, или один из них имеет значение неправильного типа (например, символьное вместо числового или наоборот).

R Tape loading error — ошибка при загрузке с ленты

неудачно прошла процедура загрузки, подгрузки или проверки файла.

Символы ZX Spectrum*

Таблица 1.

Код	Симв.	Код	Симв.	Код	Симв.	Код	Симв.	Код	Симв.	Код	Симв.
32	про-бел	53	5	75	K	97	a	119	w	141	▟
		54	6	76	L	98	b	120	x	142	▙
33	!	55	7	77	M	99	c	121	y	143	█
34	"	56	8	78	N	100	d	122	z	144	[A]
35	#	57	9	79	O	101	e	123	{	145	[B]
36	$	58	:	80	P	102	f	124	|	146	[C]
37	%	59	;	81	Q	103	g	125	}	147	[D]
38	&	60	<	82	R	104	h	126	~	148	[E]
39	'	61	=	83	S	105	i	127	©	149	[F]
40	(	62	>	84	T	106	j	128	□	150	[G]
41	)	63	?	85	U	107	k	129	▝	151	[H]
42	*	64	@	86	V	108	l	130	▘	152	[I]
43	+	65	A	87	W	109	m	131	▀	153	[J]
44	,	66	B	88	X	110	n	132	▗	154	[K]
45	-	67	C	89	Y	111	o	133	▐	155	[L]
46	.	68	D	90	Z	112	p	134	▚	156	[M]
47	/	69	E	91	[	113	q	135	▜	157	[N]
48	0	70	F	92	/	114	r	136	▖	158	[O]
49	1	71	G	93	]	115	s	137	▞	159	[P]
50	2	72	H	94	↑	116	t	138	▌	160	[Q]
51	3	73	I	95	_	117	u	139	▛	161	[R]
52	4	74	J	96	£	118	v	140	▄	162	[S]

* Коды 0...31 управляют выводом информации на экран и называются контрольными (см. табл. 2).

Код	Симв.	Код	Симв.	Код	Симв.	Код	Симв.
163	[T]	187	SQR	211	OPEN #	235	FOR
164	[U]	188	SGN	212	CLOSE #	236	GO TO
165	RND	189	ABS	213	MERGE	237	GO SUB
166	INKEY$	190	PEEK	214	VERIFY	238	INPUT
167	PI	191	IN	215	BEEP	239	LOAD
168	FN	192	USR	216	CIRCLE	240	LIST
169	POINT	193	STR$	217	INK	242	PAUSE
170	SCREEN$	194	CHR$	218	PAPER	243	NEXT
171	ATTR	195	NOT	220	BRIGHT	244	POKE
172	AT	196	BIN	221	INVERSE	245	PRINT
173	TAB	198	AND	222	OVER	246	PLOT
174	VAL$	199	<=	223	OUT	247	RUN
176	VAL	200	>=	224	LPRINT	248	SAVE
177	LEN	201	<>	225	LLIST	249	RANDOMIZE
178	SIN	202	LINE	226	STOP	250	IF
179	COS	203	THEN	227	READ	251	CLS
180	TAN	204	TO	228	DATA	252	DRAW
181	ASN	205	STEP	229	RESTORE	253	CLEAR
182	ACS	206	DEF FN	230	NEW	254	RETURN
183	ATN	207	CAT	231	BORDER	255	COPY
184	LN	208	FORMAT	232	CONTINUE		
185	EXP	209	MOVE	233	DIM		
186	INT	210	ERASE	234	REM		

Контрольные коды ZX Spectrum

Таблица 2.

Код	Набор на клавиатуре	Действие
0	CS/SS + CS/8	Не используется
1	CS/SS + CS/9	Не используется
2	CS/SS + 8	Не используется
3	CS/SS + 9	Не используется

Код	Набор на клавиатуре	Действие
4	CS/3	Не используется
5	CS/4	Не используется
6	CS/2	Установка позиции печати на середину экрана (аналогично запятой в PRINT)
7	CS/1	Не используется
8	CS/5	Перемещение позиции печати на одно знакоместо влево (Backspace)
9	CS/8	Не используется
10	CS/6	Не используется
11	CS/7	Не используется
12	CS/0	Не используется
13	Enter	Возврат каретки + перевод строки (Carriage Return + Line Feed)
14	CS/SS	Не используется
15	CS/9	Не используется
16	CS/SS + 0	Управление цветом тона (INK control)
17	CS/SS + 1	Управление цветом фона (PAPER control)
18	CS/SS + 2	Управление мерцанием (FLASH control)
19	CS/SS + 3	Управление яркостью (BRIGHT control)
20	CS/SS + 4	Управление инверсией (INVERSE control)
21	CS/SS + 5	Управление наложением (OVER control)
22	CS/SS + 6	Управление позицией печати (AT control)
23	CS/SS + 7	Управление табуляций (TAB control)
24	CS/SS + CS/0	Не используется
25	CS/SS + CS/1	Не используется
26	CS/SS + CS/2	Не используется
27	CS/SS + CS/3	Не используется
28	CS/SS + CS/4	Не используется
29	CS/SS + CS/5	Не используется
30	CS/SS + CS/6	Не используется
31	CS/SS + CS/7	Не используется

КОМПИЛЯТОРЫ SPECTRUM-БЕЙСИКА

В начале книги, помимо интерпретации, упоминался еще один способ перевода программ с языков высокого уровня на язык кодов процессора — компиляция. Главное преимущество этого способа — увеличение быстродействия программ.

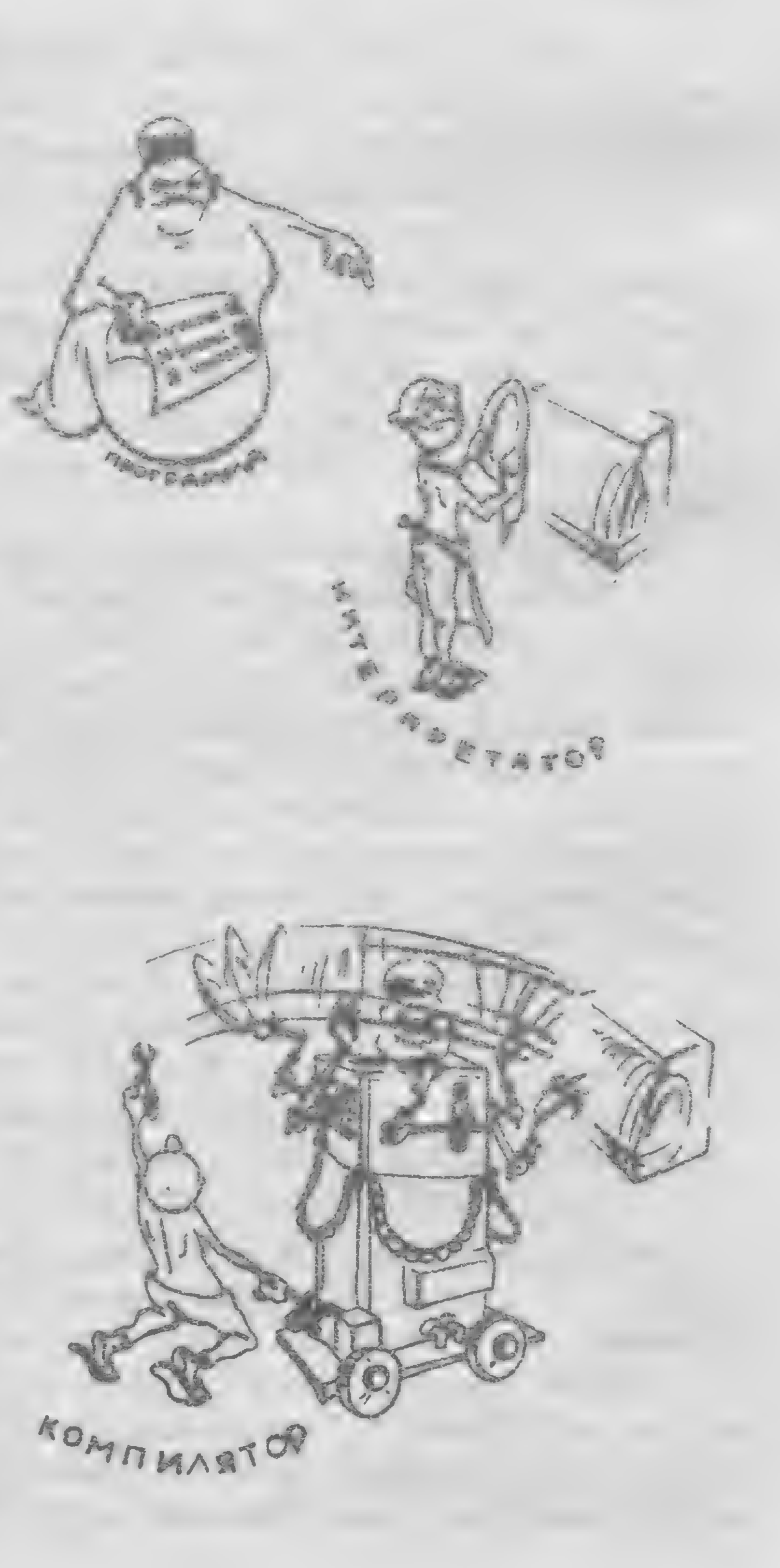
ZX Spectrum может порадовать программистов тем, что для него создано множество различных компиляторов Бейсика, от самых простых целочисленных, не «переваривающих» дробные числа и понимающих лишь ограниченный набор ключевых слов, до мощных, которые, кроме поддержки почти всех операторов базового интерпретатора, реализуют еще и некоторые дополнительные инструкции.

Но, к сожалению, компиляторы Spectrum-Бейсика не удовлетворяют всем требованиям, предъявляемым к подобного типа программам. Например, ни один из них не поддерживает полный набор инструкций интерпретатора. Кроме того, (и это, пожалуй, самое неприятное) оттранслированная программа не работает без присутствия в памяти компьютера самого компилятора. Дело в том, что любой компилятор содержит так называемые библиотеки системных процедур — набор подпрограмм в машинных кодах. Компиляторы, реализованные на компьютерах более высокого класса, как правило, вставляют в оттранслированный текст лишь те системные процедуры, которые нужны именно в данной программе. Авторы же компиляторов Spectrum-Бейсика решили не усложнять себе задачу, и в результате написанные ими продукты «навешивают» на оттранслированную программу все, без выбора, системные процедуры, а заодно и саму процедуру компиляции. А от такого упрощения — лишь дополнительный расход памяти.

Теперь коротко о том, какие компиляторы и почему будут описаны в этой главе. Мы решили ограничиться программами с безупречной репутацией. Именно поэтому в книгу не включено описание пакета Toolkit/Blast, склонного (в том виде, в каком он дошел до нашего Отечества) к непредсказуемому поведению, не соответствующему его блестящей аттестации в фирменном описании. Еще мы старались отобрать достаточно непохожие друг

на друга компиляторы, занимающие свою собственную «экологическую нишу». Так, целочисленный ZX-Compiler v.1.0 — самый простой и наименее совместимый по синтаксису с интерпретатором, но он же и самый короткий. MCoder 2 занимает в памяти почти в два раза больше места, но, в отличие от ZX-Compiler, понимает, что такое числовые массивы. Softek IS массивы не «переваривает», но поддерживает несколько новых инструкций, неведомых интерпретатору. Softek FP является простейшим из компиляторов, умеющим правильно делить один на два, а Tobos FP наиболее близок по синтаксису к интерпретатору (список ограничений минимален), но считает дробные числа с точностью не до восьми, а лишь до семи значащих цифр. Так что имеет смысл познакомиться с каждым из перечисленных компиляторов, чтобы в конкретном случае найти оптимальный компромисс между скоростью работы скомпилированной программы и ее возможностями.

И все же, несмотря на различия, описанные в книге компиляторы имеют немало общих черт. Но текст описаний не выиграл бы от многочисленных перекрестных ссылок от одного компилятора на другой. Поэтому мы старались (а удалось ли — судить читателю) найти золотую середину между требованием автономности каждого описания и стремлением не повторяться.



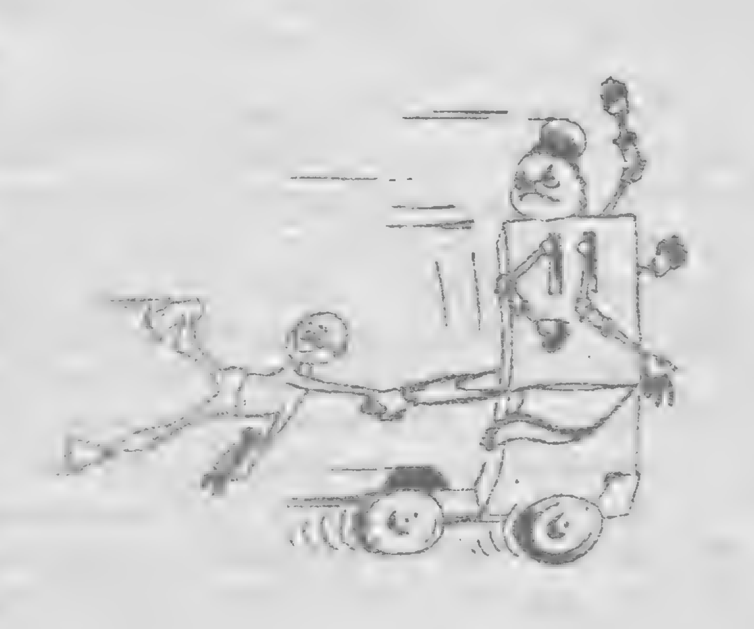
Несколько слов о структуре этой главы. Сначала будут рассмотрены три целочисленных компилятора, а затем еще два, поддерживающие работу с дробными числами. Специальный раздел посвящен записи готового программного продукта на магнитный носитель. Справочный материал, касающийся совместимости синтаксиса компиляторов и интерпретатора, приведен в Приложении 1. Основные «тактико-технические» характеристики рассматриваемых компиляторов содержит Приложение 2.

Разные компиляторы могут иметь одинаковые или похожие названия, и, наоборот, одна и та же программа может встречаться под разными именами. Поэтому при рассмотрении очередного компилятора будут

оговариваться его отличительные особенности, а также, по возможности, перечисляться различные имена, под которыми он встречается.

Компилятор диалекта Laser-Basic рассмотрен в соответствующей главе. Впрочем, он может применяться и для компиляции обычных бейсик-программ.

Последнее замечание адресовано владельцам дисковой операционной системы TR-DOS. Все рассматриваемые компиляторы не поддерживают инструкции обращения к дисководу. Существующие «дисковые» версии компиляторов умеют лишь загружаться с диска.



ЦЕЛОЧИСЛЕННЫЕ КОМПИЛЯТОРЫ

ZX-Compiler

ZX-Compiler v.1.0* — один из первых компиляторов Spectrum-Бейсика, почти ровесник Speccy. Он также известен под именем Integer Compiler, или просто Compiler.

После загрузки программа предложит нажать любую клавишу, и если последовать ее совету, то автоматически выполнится оператор NEW: картинка пропадет, и на экране появится строчка

© 1982 Sinclair Research Ltd.

Эта особенность загрузки ZX-Compiler обычно вводит в заблуждение неопытных программистов, которые воспринимают ее как сброс компьютера. В действительности же все идет по плану: область бейсик-программы очистилась от загрузчика, а собственно компилятор остался в неприкосновенности**.

Оператор NEW можно выполнять и самостоятельно для быстрого удаления текста исходной бейсик-программы. При этом и компилятор, и скомпилированная программа остаются нетронутыми.

---

* Фирма Threlfall & Hodgson, 1982 г. Файлы: COMPILER (тип BASIC, длина 875 байт) и COMPILER (тип CODE, длина 3600 байт, адрес загрузки 59990).

** Так ведут себя после загрузки многие компиляторы Spectrum-Бейсика.

После загрузки компилятор с пакетом рабочих процедур располагается в памяти с адреса 59990 по 63590. Содержимое этой области изменять нельзя.

Исходный текст программы на Бейсике вводится обычными способами: с клавиатуры, с ленты, с диска. Бейсик-программу можно редактировать, запускать на исполнение и т. п. Присутствие в памяти компилятора никак не отражается на работе интерпретатора Spectrum-Бейсика, если не считать некоторого уменьшения объема доступной ему памяти.

Объемы исходного бейсиковского текста и скомпилированной программы задаются установкой RAMTOP. Напомним, что эта системная переменная ограничивает «сверху» область, предназначенную для бейсик-программы. С другой стороны, значение RAMTOP, увеличенное на единицу, определяет адрес, с которого после компиляции будет размещена скомпилированная программа (см. табл. 3 на стр. 121). Заранее (то есть до компиляции) трудно подобрать оптимальное расположение RAMTOP, так как предсказать в точности размер скомпилированной программы практически невозможно. Обычно он близок к размеру исходного текста, несколько превышая его. Поэтому первоначально ZX-Compiler устанавливает RAMTOP на адрес 39999. При необходимости значение RAMTOP в любой момент можно изменить обычным способом, введя с клавиатуры оператор:

```
CLEAR ADDR
```

– где ADDR — требуемое значение RAMTOP.

При завышенном значении RAMTOP может возникнуть ситуация, когда оттранслированной программе не будет хватать отведенного для нее места. В этом случае компилятор прервет работу и выдаст сообщение:

```
Out of memory
```

Попробуем скомпилировать бейсик-программу с большим временем выполнения, например:

```
10 CLS
20 FOR A=0 TO 175
30 FOR B=0 TO 255
40 PLOT B, A
50 NEXT B: NEXT A
```

Эта программа последовательно, точка за точкой, закрашивает экран цветом тона. Нужно запастись достаточным терпением, чтобы, запустив ее оператором RUN, дождаться сообщения 0 OK. Прибегнем к помощи компилятора.

Запускается ZX-Compiler оператором

```
RANDOMIZE USR 60000
```

Компиляция осуществляется в два прохода. Второй проход нужен для того, чтобы правильно расставить адреса переходов и зарезервировать память для хранения переменных скомпилированной программы (последние, как и переменные бейсик-программ, хранятся сразу после основного текста). В процессе компи-

ляции в исходном тексте могут быть обнаружены синтаксические ошибки: наличие инструкций, законных с точки зрения интерпретатора, но не поддерживаемых компилятором. Ошибки отлавливаются, как правило, на первом проходе, при этом компиляция останавливается и на экран выводится часть «дефектной» строки до «ошибочного» оператора, помеченного знаком вопроса.

В процессе каждого прохода на экран выводится текст исходной программы. После успешного завершения первого прохода необходимо нажать любую клавишу (кроме **Space**). Нажатие **Space** или **Break** (**CS/Space**) прервет процесс, и компьютер перейдет в режим интерпретатора.

Компиляция сопровождается также отображением в верхней части экрана адреса окончания скомпилированной на данный момент части программы. Это число достигает своего максимального значения в конце второго прохода.

Итак, компиляция нашего примера успешно завершилась сообщением 0 OK.

По адресу конца скомпилированной программы (для рассматриваемого примера он равен 40123) можно рассчитать ее длину:

40123–(RAMTOP+1)

Подставив значение RAMTOP, равное 39999, получим число 123. Для сравнения, исходная программа на Бейсике вместе с переменными занимает 117 байт.

Запускается скомпилированная программа оператором

```
RANDOMIZE USR (RAMTOP+1)
```

Для запуска нашего примера введем с клавиатуры

```
RANDOMIZE USR 40000
```

Не правда ли, впечатляющее зрелище? Программа, на выполнение которой интерпретатору требовалось почти 6 минут, в скомпилированном виде выполняется за 22 секунды!

Да, действительно, целочисленные компиляторы, к которым относится ZX-Compiler, обеспечивают поистине фантастическую скорость выполнения операций. Кроме того, приведенный пример не может в полной мере продемонстрировать эффективность компилятора, так как скорость выполнения программ, использующих вывод на экран графики, сильно ограничена быстродействием подпрограмм ПЗУ. Трансляция программ, выполняющих только арифметические операции, дает гораздо больший выигрыш во времени. Например, следующая программа:

```
10 LET C=0
20 FOR A=0 TO 175
30 FOR B=0 TO 255
40 LET C=C+1
50 NEXT B: NEXT A
```

после компиляции будет работать уже более чем в 80 раз быстрее оригинала!

Однако за такую высокую скорость приходится дорого расплачиваться: ZX-Compiler предъявляет довольно жесткие требования к исходному тексту программы.

После включения в текст программы, например, оператора DIM A(10), на первом же проходе компилятор с недовольным гудением остановит трансляцию и выдаст сообщение Nonsense in BASIC, указав место ошибки мигающим знаком ?. Программный курсор в этом случае расположится в строке, содержащей ошибку, так что для вызова ее на редактирование достаточно будет нажать **Edit (CS/1)**.

Существует ряд корректных с точки зрения стандартного Бейсика операторов и функций, компиляцию которых ZX-Compiler не поддерживает. Их исчерпывающий перечень приведен в Приложении 1. Здесь лишь пробежимся по этому списку.

Разумеется, не поддерживаются функции, возвращающие, в общем случае, нецелочисленный результат (SIN, COS, TAN, ASN, ACS, ATN, LN, EXP), а также вызов числа PI. Исключением является функция SQR, но при ее расчете отбрасывается дробная часть результата. То же самое происходит и при выполнении операции деления. Не допускается использование в исходном тексте операции возведения в степень и функций, определяемых пользователем (DEF FN, FN).

Не поддерживается работа с символьными переменными, и, следовательно, нельзя включать в текст бейсик-программы функции VAL, VAL$, SCREEN$, LEN, STR$ и оператор INPUT с ключевым словом LINE. Опрос клавиатуры с помощью функции INKEY$ в таком случае возможно реализовать только «в тандеме» с функцией CODE, например:

```
LET c = CODE INKEY$
PRINT CODE INKEY$
```

Не допускается работа с массивами, как числовыми, так и символьными, вне зависимости от их размерности.

Запрещено применение операторов GO TO и GO SUB с переменной или выражением в качестве параметров. Номер строки, к которой осуществляется переход, можно указывать только в явном виде — числом.

Выражения, используемые в качестве аргументов функций, во всех случаях должны быть заключены в скобки, например:

```
CIRCLE X, (Y+16), 10
```

или

```
PRINT AT (A+1), (2*8)
```

Не поддерживаются логические операции (NOT, OR, AND).

Функция RND генерирует значения в диапазоне от 0 до 32767.

Оператор STOP, независимо от его местонахождения в программе, трактуется как ограничитель текста программы. Весь текст после этого оператора при компиляции игнорируется.

Запрещены операторы SAVE, LOAD, CONTINUE, CLEAR, LIST, NEW, RANDOMIZE.

Результат функции USR с символьным аргументом выдается в скомпилированной программе в так называемом «дополнительном

коде» в диапазоне от −32768 до +32767*. Например, оператор PRINT USR "A" выводит на первый взгляд странный результат: −168 (вместо привычного 65368). Для чисел в диапазоне от 0 до 32767 дополнительное представление числа совпадает с обычным. Для чисел в диапазоне от 32768 до 65535 пересчет из одной формы в другую следует производить по формулам:

N1=N2−65536 (N2=32767...65535)
N2=N1+65536 (N1=−32768...0)

— где N1 — дополнительное представление числа N2. Функция USR с символьным аргументом — это единственный случай, когда приходится помнить об этом, поскольку для инструкций POKE и PEEK, а также для USR с числовым аргументом допустимо указывать значения аргументов как в обычном, так и в дополнительном представлении.

Операторы READ и DATA работают без особенностей, однако RESTORE употребляется только с параметром.

Все компиляторы Spectrum-Бейсика позволяют включать в текст компилируемой программы обращения к подпрограммам в машинных кодах, осуществляемые с помощью функции USR. Однако поскольку ZX-Compiler не поддерживает оператор RANDOMIZE, а также не допускает использования выражений в качестве параметров операторов GO TO и GO SUB, наиболее часто употребляющиеся формы RANDOMIZE USR <адрес>, GO TO USR <адрес> и GO SUB USR <адрес> трактуются как ошибочные. Приходится использовать другие операторы, например:

```
PRINT USR <адрес>
LET N=USR <адрес>
```

Здесь нелишне напомнить, что функция USR при использовании ее для обращения к процедурам в машинных кодах возвращает интерпретатору содержимое регистровой пары BC, которое воспринимается как параметр оператора, предшествовавшего USR. Так, оператор LET N=USR <адрес> присвоит переменной N значение, содержащееся в регистровой паре BC, оператор PRINT USR выведет его на экран и т. п.

Несмотря на то, что ZX-Compiler по природе своей может работать только с целыми числами, все же для параметров оператора BEEP он делает исключение. Ведь использование BEEP теряет смысл, если, например, длительность звука нельзя сделать меньше секунды. Поэтому ZX-Compiler позволяет задавать параметры BEEP в виде натуральных дробей, например, так:

```
BEEP 1/10, (n+1/8)
```

У читателя может возникнуть вопрос: «Стоит ли вообще пользоваться таким компилятором, как ZX-Compiler?» Действительно, нужен ли этот самый старый, самый «ограниченный» и не самый быстродействующий компилятор? Все перечисленные недостатки

* Это происходит потому, что старший бит двухбайтового числа трактуется не как значащий разряд, а как знак: 0 — плюс, 1 — минус.

можно простить за одно лишь достоинство ZX-Compiler, которое в некоторых случаях может оказаться решающим: он занимает в памяти почти в два раза меньше места (всего 3600 байт), чем наименьший из других компиляторов, описываемых в этой книге.

О том, как записать скомпилированную программу на внешний носитель и снабдить ее программой-загрузчиком, то есть сделать из нее законченный программный продукт, будет сказано в разделе, специально посвященном этому вопросу (см. стр. 120).

MCoder 2

Компилятор MCoder 2* — вторая версия ZX-Compiler, значительно превосходящая своего предшественника. Основным и неоспоримым достоинством MCoder 2 является возможность работы с одномерными числовыми массивами и символьными переменными. В остальном список особенностей компилятора почти совпадает с соответствующим списком ZX-Compiler (подробно — в Приложении 1):

- не поддерживаются математические функции SIN, COS, TAN, ASN, ACS, ATN, LN, EXP, PI, функции VAL, VAL$ и функции, определяемые пользователем (DEF FN, FN);
- недопустимо использование переменных или выражений в операторах GO TO и GO SUB;
- допускаются все варианты применения функции USR для вызова внешних процедур в машинных кодах, кроме GO TO USR <адрес> и GO SUB USR <адрес>;
- выражения, используемые в качестве параметров операторов PLOT, DRAW и CIRCLE должны быть заключены в круглые скобки;
- не поддерживаются логические операции (NOT, AND**, OR);
- оператор STOP рассматривается как ограничитель текста программы, и весь последующий текст игнорируется;
- не поддерживаются операторы SAVE, LOAD, CONTINUE, LIST, NEW, а также CLEAR с параметром;
- операторы READ и DATA работают, как в стандартном Бейсике, однако RESTORE не может употребляться без параметра;
- в параметрах BEEP можно использовать натуральные дроби.

После загрузки компилятора RAMTOP устанавливается на адрес 39999. В процессе работы значение RAMTOP можно изменять оператором CLEAR <новое значение>. Скомпилированная программа размещается с адреса RAMTOP+1.

Компиляция запускается оператором RANDOMIZE USR 60000 и осуществляется в два прохода. Дополнительным удобством является то, что по окончании работы компилятор выводит на экран информацию о длине скомпилированного кода.

---

* Фирма Threlfall & Hodgson, 1983 г. Файлы: MCODER2 (тип BASIC, длина 870 байт) и MCODER2 (тип CODE, длина 5375 байт, адрес загрузки 59990).

** Может использоваться только в качестве условия в операторе IF...THEN.

Скомпилированная программа запускается оператором

```
RANDOMIZE USR (RAMTOP+1)
```

MCoder 2 позволяет ускорить работу программ обычно в 60...90 раз, что несколько превосходит показатели ZX-Compiler v.1.0. Лучшие результаты получаются при использовании в программе только арифметических действий, без вывода графики.

Softek IS

Компилятор Softek IS* по сравнению с ZX-Compiler не только накладывает на текст исходной бейсик-программы меньше ограничений, но и добавляет несколько новых инструкций, отсутствующих в Spectrum-Бейсике. Кроме того, он значительно увеличивает скорость выполнения скомпилированной программы. Тест-программа:

```
10 LET C=0
20 FOR A=0 TO 175
30 FOR B=0 TO 255
40 LET C=C+1
50 NEXT B: NEXT A
```

после обработки компилятором Softek IS ускоряет свою работу более чем в 95 раз. Правда, за это пришлось заплатить почти двукратным (по сравнению с ZX-Compiler v.1.0) увеличением объема самого компилятора, который составил 6 килобайт.

Процедура запуска Softek IS выполнена несколько иначе, чем ZX-Compiler. После окончания загрузки компилятор запрашивает адрес, по которому требуется разместить скомпилированную программу:

```
RAMTOP at 40000? (Y or N)
```

После утвердительного ответа значение системной переменной RAMTOP устанавливается равным 40000, выполняется оператор NEW и управление передается интерпретатору Бейсика. При отрицательном ответе программа запрашивает новое значение RAMTOP, ввод которого завершается клавишей **Enter**. Значение RAMTOP, а следовательно, и адрес размещения скомпилированной программы можно изменить в процессе работы оператором CLEAR (более подробно об этом см. стр. 106).

Ввод программы, предназначенной для компиляции, ее редактирование и отладка ничем не отличаются от обычной работы в Spectrum-Бейсике.

Запускается компиляция оператором

```
RANDOMIZE USR 59300
```

---

* Автор Martin Lewis, фирма Softek, 1983 г. Файлы: SOFTEK IS (тип BASIC, длина 620 байт) и COMET (CODE, длина 6000 байт, адрес загрузки 59300).

В процессе компиляции на экране отображаются: адреса начала и конца размещения скомпилированной программы, адрес окончания области переменных скомпилированной программы и — в специальном окне — сообщения компилятора.

Компиляция осуществляется в два прохода, сопровождающихся, соответственно, сообщениями:

FIRST PASS
SECOND PASS

После успешного завершения компиляции на экран выдается привычное

0 OK

и текст

NO ERRORS

В случае обнаружения недопустимой для Softek IS конструкции компиляция прерывается с сообщением ERROR и на экран выводится участок программы, непосредственно предшествующий ошибке. Место сбоя отмечается мигающим знаком ?. Как и в ZX-Compiler, программный курсор автоматически устанавливается на «дефектную» строку программы.

Ограничения в использовании операторов и функций Spectrum-Бейсика частично совпадают с ограничениями, накладываемыми ZX-Compiler (но есть и отличия):

- не поддерживается работа с массивами;
- допустимы все формы обращения к внешним подпрограммам в машинных кодах, кроме GO TO USR <адрес> и GO SUB USR <адрес>;
- все ограничения, касающиеся использования функций при работе с ZX-Compiler, справедливы и для Softek IS: не поддерживаются математические функции SIN, COS, TAN, ASN, ACS, ATN, PI, LN, EXP и, в отличие от ZX-Compiler, SQR, а также VAL, VAL$, LINE и функции, определяемые пользователем;
- допустимы операции с символьными переменными, в связи с чем возможно использование функций STR$, LEN и SCREEN$, а также INKEY$;
- не поддерживаются операторы MERGE, CONTINUE, LIST, NEW;
- использование SAVE, LOAD, VERIFY допустимо только при операциях с файлами типа CODE;
- оператор CLEAR может быть использован в программе только без параметра;
- выражения, используемые в качестве параметров операторов, не обязательно заключать в скобки;
- допустимы логические операции NOT, AND, OR, однако многоуровневое их вложение может привести к ошибке;
- операторы READ и DATA выполняются, но требуют наличия в программе хотя бы одного оператора RESTORE (с аргументом или без него), предшествующего первому оператору READ.

Дополнительные операторы, распознаваемые Softek IS, но отсутствующие в Spectrum-Бейсике, вводятся в текст исходной прог-

раммы вслед за оператором REM. Они составляются из прописной буквы (идентификатора оператора) и списка параметров.

REM B

останавливает работу скомпилированной программы, если в момент его выполнения нажата клавиша **Break** (**CS/Space**).

REM S, A, x, y

выводит символ на экран с точностью до пикселя. Параметры: A — код символа, x (0...175) и y (0...255) — расстояния в пикселях от левого нижнего угла основного экрана. При выходе за пределы экрана печать продолжается с его противоположной стороны. Все параметры могут быть переменными или выражениями.

Приведем пример вывода на экран символьной переменной:

```
10 LET B=80
20 LET W$="Softek IS v1.2"
30 FOR A=1 TO 14
40 LET B=B+1
50 REM S, CODE W$(A), A*7, B
60 NEXT A
```

Естественно, увидеть что-нибудь на экране можно будет только после компиляции этой программы (см. рис. 6): интерпретатор воспримет REM S как REM и только.

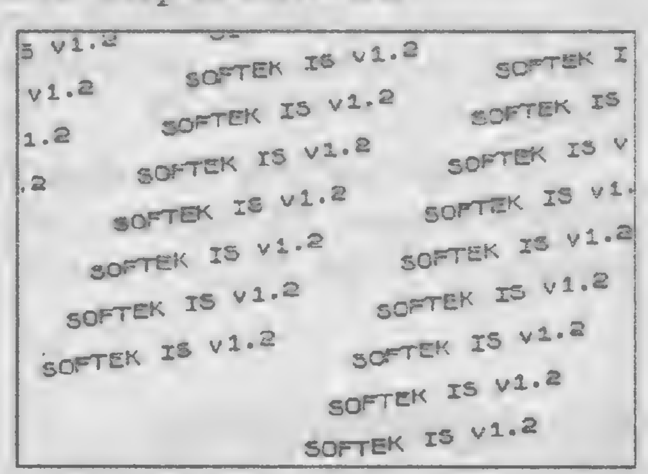


Рис. 6. Пример работы оператора REM S.

Оператор REM S может быть очень полезен при использовании набора узких символов (например, 4×8 пикселей). Тогда можно обеспечить вывод на экран более 32 символов в строке и без применения специального драйвера.

REM S имеет одну интересную особенность — он способен выводить на экран не 96 символов (коды от 32 до 127), как это принято для операторов печати Spectrum-Бейсика, а 256. Дополнительные символы хранятся в обычном формате (по 8 байт на символ) в нескольких областях памяти, положение которых определяется, исходя из значений системных переменных CHARS (23606/07) и UDG (23675/76):

- символы с кодами от 0 до 31, которым в стандарте ASCII соответствуют управляющие коды, считываются с адреса, хранящегося в CHARS, и занимают 256 байт;
- символы с кодами от 32 до 127 — с адреса (CHARS)+256 и занимают положенные им 768 байт до адреса (CHARS)+1023 включительно, как обычный символьный набор;
- символы с кодами от 143 до 255, которым в системе ZX Spectrum соответствуют ключевые слова Бейсика, располагаются в памяти с адреса (UDG)−8 по адрес (UDG)+895;
- и, наконец, символы с кодами от 128 до 142 (в обычных условиях — символы псевдографики) формируются из данных, записанных в области памяти, начиная с адреса (UDG)+1920 по (UDG)+2039.

Знание последней особенности дает возможность использовать в скомпилированных программах альтернативный символьный набор из 128 символов совместно с 128 символами UDG, один раз прибегнув к модификации системных переменных UDG и CHARS. Следует подчеркнуть, что эти особенности присущи только оператору REM S, но не относятся к стандартным операторам вывода на экран (PRINT CHR$).

REM M, A1, A2, ..., An

включает в скомпилированную программу фрагмент в машинных кодах: A1, A2 и т. д. (целые числа от 0 до 255). Для успешного возврата из процедуры в кодах она должна заканчиваться кодом 201 (RET). Оператор REM M позволяет писать наиболее критичные по времени выполнения фрагменты программы непосредственно в машинных кодах, что значительно расширяет сферу применения компилятора Softek IS.

На этом обзор целочисленных компиляторов можно считать законченным. Теперь наступило время перейти к следующей теме, а именно к так называемым компиляторам «с плавающей точкой».

КОМПИЛЯТОРЫ, РАБОТАЮЩИЕ С ВЕЩЕСТВЕННЫМИ ЧИСЛАМИ

Softek FP

Компилятор Softek FP* (Softek Floating Point), известный также под именем Full Compiler, разработан тем же автором, что и Softek IS, и внешне напоминает последний. По-видимому, изначально Softek IS и Softek FP составляли нечто вроде программного пакета, предназначенного для ускорения работы различных бейсик-программ. Причем там, где была необходима максимальная скорость, но не требовалось особо сложных математических вычислений, рекомендовалось использовать Softek IS, а в остальных случаях мог быть использован Softek FP, работающий несколько медленнее, зато позволяющий обрабатывать как целые, так и вещественные числа.

Две версии этого компилятора — Full Compiler v.1.1 и FP48K v.1.7 — не слишком сильно отличаются друг от друга. Full Compiler v.1.1 состоит из бейсик-загрузчика и файла в машинных кодах. FP48K v.1.7 состоит из трех файлов: загрузчика, фирменной заставки Softek FP и блока кодов собственно компилятора. Версия 1.1 по окончании загрузки, как и Softek IS, запра-

* Автор Martin Lewis, фирма Softek, 1983 г. Файлы версии 1.1: COMPILER (тип BASIC, длина 707 байт), MACHINE CODE (тип CODE, длина 6050 байт, адрес загрузки 59300). Файлы версии 1.7: FP48k v.1.7 (тип BASIC, длина 36 байт), <пустое имя> (тип CODE, длина 1000 байт, адрес загрузки 23500), m/CODE (тип CODE, длина 6100 байт, адрес загрузки 59300).

шивает значение RAMTOP, однако после его ввода не происходит очистки области Бейсика оператором NEW, поэтому перед началом работы необходимо сделать это вручную. Оригинальная версия 1.7 устанавливает RAMTOP на адрес 39999 без запроса и выполняет оператор NEW автоматически (в дисковой версии 1.7 ввод значения RAMTOP и выполнение NEW необходимо сделать вручную). Значение RAMTOP всегда можно изменить оператором CLEAR <адрес>, где <адрес> — число, на единицу меньшее, чем требуемый адрес размещения скомпилированной программы (подробнее смотрите в описании ZX-Compiler, стр. 106).

Работа с исходными текстами на Бейсике ничем не отличается от обычной. Необходимо только помнить, что область памяти над RAMTOP отводится под скомпилированную программу, а область с адреса 59300 по адрес 65350 для версии 1.1 или по 65400 для версии 1.7 занята самим компилятором, поэтому указанные области памяти нельзя использовать для других целей.

Обе версии запускаются оператором

```
RANDOMIZE USR 59300
```

В процессе компиляции на экране отображаются: адреса начала и конца размещения скомпилированной программы, адрес окончания области переменных и в специальном окне — сообщения компилятора. Все выводимые компилятором сообщения соответствуют аналогичным сообщениям Softek IS.

Далее о самом главном — об ограничениях, накладываемых Softek FP на текст исходной бейсик-программы:

- диапазон обрабатываемых компилятором чисел соответствует диапазону, в котором работает интерпретатор: от $2{,}8 \cdot 10^{-39}$ до $1{,}7 \cdot 10^{38}$. Точность представления чисел — до восьми значащих цифр, как и в интерпретаторе. При выходе абсолютной величины числа за пределы указанного диапазона в сторону увеличения выдается сообщение Number too big, в сторону уменьшения — числовое значение принимается за нулевое. Применение функций VAL (в версии 1.1) и VAL$ (в обеих версиях) при компиляции не воспринимается как ошибка, но во время работы скомпилированной программы вызывает сбой, и выдается сообщение Nonsense in BASIC;
- не диагностируется как ошибка, но приводит к сбою и применение оператора INPUT с ключевым словом LINE;
- не поддерживаются функции, определяемые пользователем (DEF FN и FN);
- все используемые массивы, как числовые, так и символьные, должны быть одномерными;
- не допустимы инструкции GO TO и GO SUB с нечисловыми параметрами (выражениями и переменными);
- не поддерживаются операторы MERGE, CONTINUE, LIST, NEW. Использование SAVE, LOAD, VERIFY допустимо в операциях с кодовыми файлами. Оператор STOP, встреченный в тексте программы, не воспринимается как ограничитель текста. STOP обязательно должен завершать текст бейсик-программы, если в ней хотя бы один раз использовался оператор CLEAR — в

противном случае по окончании работы скомпилированной программы произойдет сброс компьютера. Программа, остановленная оператором STOP, не может быть продолжена вводом CONTINUE;

- все сказанное об использовании логических операций при работе с Softek IS справедливо и для Softek FP;
- на применение операторов READ, DATA и RESTORE ограничений не накладывается.

Softek FP так же, как его целочисленный «собрат» Softek IS, реализует некоторые дополнительные инструкции, не известные интерпретатору Бейсика. Операторы REM B (опрос нажатия CS/Space) и REM M (включение фрагмента в машинных кодах) работают так же, как и в Softek IS (см. стр. 113). Но, к величайшему сожалению, один из самых интересных дополнительных операторов Softek IS — печать символов в произвольную точку экрана (REM S) — наотрез отказывается работать в Softek FP.

Softek FP дополняет Spectrum-Бейсик еще двумя новыми операторами: REM E и REM O.

REM E, n

указывает компилятору на строку в тексте программы, где находится подпрограмма обработки ошибок*. Находясь в произвольном месте программы, этот оператор обеспечивает при возникновении любой ошибки переход к строке с номером n. Естественно, что применение оператора REM E полностью исключает возможность останова программы при помощи оператора REM B. Более того, запущенная на выполнение программа, включающая в себя строку REM E, n, не может быть остановлена обычными способами. Даже естественное окончание программы с выводом сообщения 0 OK расценивается как ошибка и вызывает переход на строку, указанную в REM E. Это создает известные неудобства: по завершению программы невозможен возврат к работе в интерпретаторе Бейсика. Но, с другой стороны, это в какой-то степени защищает программу от несанкционированного доступа.

Для того чтобы обезопасить себя от возможных неприятностей, связанных с использованием REM E, лучше не включать этот оператор при пробных запусках программы, а вписывать его в уже окончательно отлаженный текст. Перед запуском программы с оператором REM E нужно не забыть сохранить ее исходный текст на внешнем носителе. В противном случае весь результат работы может быть потерян из-за невозможности вернуться в интерпретатор.

REM O, q, n1, n2, ..., nn

реализует оператор группового перехода. Во многих версиях Бейсика в этом случае используется ключевое слово ON, в частности, в Beta Basic он записывается так:

```
GO TO ON q; n1, n2, ..., nn
```

---

* Отметим, что этот оператор работает только в Softek FP версии 1.1.

В зависимости от значения параметра q осуществляется переход по одному из номеров строк, указанных в списке n1, n2, ..., nn. При q=1 осуществляется переход к строке с номером n1, при q=2 — к строке с номером n2 и т. д. Если значение q меньше или равно нулю, либо превышает число элементов списка, либо имеет дробное значение, то управление передается следующей за оператором REM O строке. Необходимо помнить, что q не может быть числом или выражением — только числовой переменной.

Tobos FP

Компилятор Tobos FP* (Tobos Floating Point) появился на свет в Польше в 1986 году и стал на сегодняшний день, пожалуй, наиболее популярным из всех известных бейсик-компиляторов для ZX Spectrum. Он удачно сочетает в себе относительно высокое быстродействие с максимальной совместимостью со Spectrum-Бейсиком. Tobos FP имеет ряд дополнительных возможностей, например, позволяет выбирать различные режимы компиляции в зависимости от нужд программиста и параметров исходного текста. Tobos может компилировать довольно большие программы, наиболее эффективно используя при этом память ZX Spectrum.

Компилятор состоит из бейсик-загрузчика (известно два варианта) и блока кодов длиной 12268 байт, располагающегося с адреса 53100. RAMTOP изначально устанавливается на адрес 39999, однако это значение может быть в любой момент изменено с помощью CLEAR <адрес>.

Компиляция запускается оператором

```
RANDOMIZE USR 53100
```

По завершении компиляции на экран выводится фирменная заставка Tobos FP и значения параметров, необходимых для запуска скомпилированной программы и ее записи на внешний носитель: адрес начала текста и его длина.

Tobos FP избавляет программиста от необходимости постоянно набирать последовательность операторов для запуска компилятора и скомпилированной программы. Остроумный прием позволяет полностью автоматизировать этот процесс. Необходимо только, чтобы первая строка программы содержала следующий текст:

```
1 IF 1/2=0.5 THEN RANDOMIZE USR 53100: RANDOMIZE USR ADDR:
    STOP
```

— где ADDR — адрес начала скомпилированной программы. Тогда оператор RUN, введенный с клавиатуры, вызовет компиляцию и немедленный запуск скомпилированной программы. Секрет этого приема заключается в следующем: для интерпретатора Spectrum-Бейсика величины 1/2 и 0.5 — это не совсем одно и то же, а для

---

* Авторы Jerzy Borkowski и Wojciech Skaba, 1986 г. Файлы первого варианта: TOBOS FP (тип BASIC, длина 258 байт), FLOATING POINT (тип CODE, длина 12268 байт, адрес загрузки 53100). Файлы второго варианта (заставка с бегущей строкой): TOBOS FP (тип BASIC, длина 2648 байт), файл без заголовка (тип CODE, длина 12268 байт, адрес загрузки 53100).

компилятора эти два числа равны (о причинах — чуть позже). Поэтому при запуске программы средствами интерпретатора (RUN) значение выражения 1/2–0.5 не равно нулю, и выполняются операторы, следующие за THEN, то есть осуществляются компиляция программы и ее запуск. Вычисление выражения 1/2–0.5 в скомпилированной программе даст нулевой результат, и, следовательно, управление передастся следующей строке, минуя компиляцию.

В процессе компиляции, как и во время выполнения программы, могут появляться различные сообщения об ошибках. По смыслу они, в основном, соответствуют аналогичным сообщениям интерпретатора, однако некоторые из них имеют свои специфические особенности. Так, сообщение Variable not found выдается на этапе компиляции, если в тексте программы обнаруживается обращение к необъявленной переменной или массиву. Однако при работе в среде Tobos FP может сложиться парадоксальная ситуация, когда первое обращение к переменной может предшествовать ее объявлению операторами LET, FOR, INPUT или READ. Поясним. Например, совершенно нелепый, с точки зрения интерпретатора, текст:

```
10 PRINT A
20 PRINT B
30 PRINT C
40 LET A=A+1
50 LET B=A+C
60 LET C=A+B
```

для компилятора является абсолютно нормальным. Связано это с тем, что при компиляции в памяти компьютера резервируется место для всех числовых переменных, хотя бы раз упомянутых в операторах LET, FOR, INPUT или READ, и присваивается этим переменным нулевое значение. Следовательно, результатом работы вышеприведенного примера будет вывод на экран трех нулей, так как печать переменных осуществляется до того, как им впервые присвоено значение. Однако если бы этот пример имел следующий вид:

```
10 PRINT A
20 PRINT B
30 PRINT C
```

– то на этапе компиляции возникла бы ошибочная ситуация, которая привела бы к выдаче сообщения Variable not found, так как переменные A, B и C в программе не объявлены вообще. Благодаря этому эффекту можно не заботиться о принудительном обнулении переменных перед началом работы скомпилированной программы — Tobos FP сделает это сам.

Сообщение RAMTOP no good может появиться при компиляции больших по объему исходных текстов в результате нехватки места для стека компьютера (между RAMTOP и областью бейсик-программы). В этом случае рекомендуется поднять RAMTOP приблизительно на 100 байт.

Сообщение Nonsense in BASIC может появиться при использовании не допустимых в среде Tobos FP операторов, таких как SAVE, LOAD, CONTINUE, а также CLEAR с параметром. Эти операторы относятся к числу тех немногих ограничений, которые Tobos FP все же накладывает на исходный текст программы.

Необходимо отметить, что Tobos FP не пользуется подпрограммами калькулятора ZX Spectrum, а имеет свой собственный, с другим форматом хранения чисел. Этот калькулятор производит вычисления с точностью до 7 значащих цифр, а не до 8, как это делает интерпретатор. Именно поэтому для Tobos FP величина 1/2 в точности равна 0,5. С наличием собственного калькулятора связано существенное увеличение объема Tobos FP по сравнению с компиляторами Softek.

Сообщение Out of memory возникает всякий раз, когда компилятору не хватает памяти для размещения скомпилированной программы. Эту ситуацию следует отличать от случая, когда аналогичное сообщение возникает до компиляции и означает, что компьютеру не хватает места для размещения текста исходной бейсик-программы. Для компиляции больших программ (объемом до 27 килобайт) существует ряд специальных приемов, которые будут подробно рассмотрены дальше.

Все сообщения об ошибках, выводимые не только в процессе компиляции, но и при выполнении скомпилированной программы, содержат указание на номер строки исходного текста, в которой возникла ошибочная ситуация. Это во многом облегчает поиск ошибок.

Компилятор Tobos FP представляет собой достаточно гибкое программное средство, позволяющее выбирать различные режимы компиляции в зависимости от условий. Стандартный режим компиляции, задаваемый по умолчанию, более всего удобен для предварительной отладки небольших (до 13 килобайт) программ. Для готовых отлаженных программ удобно применять режим трансляции с сокращенным форматом вывода сообщений об ошибках (без указания номера ошибочной строки в исходном тексте). Это позволяет несколько сократить объем скомпилированной программы и ускорить ее работу. Режим включается оператором POKE 53252,0, а выключается — POKE 53252,55.

Для обработки больших программ может оказаться полезным режим компиляции с уничтожением исходного текста и размещением на его месте кодов скомпилированной программы. Включается режим оператором POKE 53240,0, а выключается — POKE 53240,55.

Кроме того, при компиляции очень больших программ может помочь справиться с неизбежными проблемами рациональное распределение памяти компьютера. Для того чтобы оптимально использовать память, необходимо знать несколько собственных системных переменных Tobos FP.

Ячейки 58112/13 содержат адрес размещения скомпилированной программы для режима компиляции без уничтожения исходного текста программы. Если обе ячейки содержат нули, то код размещается традиционно по адресу RAMTOP+1, если же нет, то по адресу PEEK 58112+256*PEEK 58113.

Несложная программа на Бейсике рассчитывает по заданному адресу значения параметров операторов POKE и заносит их в системную переменную:

```
10 INPUT "ADDR", ADDR: REM Ввод адреса начала скомпилирован-
   ной программы
20 RANDOMIZE ADDR: REM Преобразование двухбайтового числа
   в два однобайтовых
30 POKE 58112, PEEK 23670
40 POKE 58115, PEEK 23671: REM Помещение полученных значений
   в системные переменные Tobos FP
50 POKE 53240,55: REM Компиляция без уничтожения текста
```

Ячейки 55631/32 содержат адрес размещения скомпилированной программы для режима трансляции с уничтожением исходного текста. Изначально эти ячейки содержат адрес 24000 для магнитофонной версии компилятора и 24100 для его дисковой версии.

Программа, автоматически помещающая требуемые значения в эту системную переменную, может выглядеть так:

```
10 INPUT "ADDR", ADDR
20 RANDOMIZE ADDR
30 POKE 55631, PEEK 23670
40 POKE 55632, PEEK 23671
50 POKE 53240, 0: REM Компиляция с уничтожением текста
```

Основные приемы компилирования больших программ сводятся, главным образом, к умелому манипулированию значениями RAMTOP и вышеперечисленных переменных. Можно порекомендовать такую последовательность работы:

1. установить RAMTOP равным 53099, освободив максимум пространства для исходного текста программы;
2. выбрать режим компиляции с уничтожением исходного текста программы оператором POKE 53240,0 и режим сокращенных сообщений об ошибках оператором POKE 53252,0;
3. запустить компиляцию оператором RANDOMIZE USR 53100 или при помощи запускающей строки и оператора RUN;
4. по окончании компиляции переместить RAMTOP на 1 байт ниже начала скомпилированной программы, выполнив CLEAR ADDR–1, и запустить последнюю на выполнение оператором RANDOMIZE USR ADDR.

Способы записи скомпилированной программы на внешний носитель подробно описаны в следующем разделе.

ЗАПИСЬ СКОМПИЛИРОВАННЫХ ПРОГРАММ

Полученный в результате компиляции код не является законченным программным продуктом, так как для его работы необходимо присутствие в памяти компьютера самого бейсик-компилятора.

Существует несколько способов записи на внешний носитель скомпилированной программы в формате, пригодном для ее самостоятельного применения. Суть этих способов сводится к написанию по окончании компиляции специальной программы-загрузчика на Бейсике, которая автоматически загружает в память компилятор, скомпилированную программу и запускает ее на выполнение. Нередко программа-загрузчик содержит в себе так называемый «самоспасатель», позволяющий после окончания компиляции одним оператором RUN создать на ленте или на диске все необходимые файлы, составляющие законченный программный продукт.

Рассмотрим несколько методов создания из скомпилированного полуфабриката законченного программного продукта.

Метод первый

Суть метода — создание на ленте или диске трех файлов: загрузчика на Бейсике, файла в машинных кодах, содержащего пакет рабочих процедур компилятора, и файла скомпилированной программы (табл. 3).

Таблица 3. **Распределение памяти при работе с компиляторами.**

P_RAMT (23732)*	
	Символы, определяемые пользователем
UDG (23675)	
	Свободная область
ADDRCOMP+LENCOMP	
	Область, занятая компилятором
ADDRCOMP	
	Рабочая область компилятора
ADDRCOMP–WSPACE	
	Свободная область
ENDVARS	
	Область переменных скомпилированной программы
ENDPROG	
	Скомпилированная программа
RAMTOP (23730)	
	...
E_LINE (23641)	
	Переменные Бейсика
VARS (23627)	
	Бейсик-программа
PROG (23635)	

* В скобках приведены адреса системных переменных, в которых записаны значения адресов соответствующих границ областей памяти.

Для осуществления замысла необходимо произвести следующие действия:

1. загрузить компилятор;
2. набрать или загрузить с внешнего носителя исходный текст программы на Бейсике;
3. произвести компиляцию;
4. освободить область Бейсика от исходной программы, выполнив оператор NEW;
5. написать загрузчик на Бейсике по следующей схеме (для работы с диском здесь и далее изменения обычные):

   10 CLEAR ADDR–1: REM *Установка RAMTOP*

   20 LOAD "COMPILER" CODE: REM *Загрузка компилятора*

   30 LOAD "NAME" CODE: REM *Загрузка скомпилированной программы*

   40 RANDOMIZE USR ADDR: REM *Запуск скомпилированной программы*
6. записать файлы на внешний носитель строкой операторов:

   SAVE "NAME" LINE 10 : SAVE "COMPILER" CODE ADDRCOMP, LENCOMP : SAVE "NAME" CODE ADDR, ENDVARS–ADDR

   – где ADDRCOMP — адрес загрузки компилятора (он не всегда совпадает с адресом его запуска), LENCOMP — длина компилятора, ADDR — адрес начала скомпилированной программы (указывается самим компилятором и равен практически всегда RAMTOP–1), ENDVARS — адрес окончания области переменных скомпилированной программы (также выводится самим компилятором). Значения ADDRCOMP и LENCOMP для различных компиляторов сведены в табл. 4. Некоторые компиляторы, например, MCoder 2 или Tobos FP вместо адреса окончания скомпилированной программы выводят ее длину, в таком случае это значение прямо подставляется в оператор SAVE вместо выражения ENDVARS–ADDR.

Таблица 4. **Параметры компиляторов, необходимые для записи готового программного продукта.**

Компилятор	ADDRCOMP	LENCOMP	Адрес запуска компилятора
ZX-Compiler 1.0	59990	3600	60000
Softek IS	59300	6000	59300
MCoder 2	59990	5375	60000
Softek FP 1.1	59300	6050	59300
Softek FP 1.7	59300	6100	59300
Tobos FP	53100	12268	53100

Приверженцам хорошего стиля, а также тем, кто не любит набирать длинные последовательности операторов на клавиатуре, рекомендуем поручить весь процесс записи программного продукта на внешний носитель самому компьютеру. Для этого рекомендуем такую программу:

```
10 SAVE "NAME" LINE 50: REM Начало «самоспасателя»
20 SAVE "COMPILER" CODE ADDRCOMP, LENCOMP
30 SAVE "NAME" CODE ADDR, ENDVARS—ADDR
40 STOP: REM Окончание «самоспасателя»
50 CLEAR ADDR—1
60 LOAD "COMPILER" CODE
70 LOAD "NAME" CODE
80 RANDOMIZE USR ADDR
```

После набора этого варианта загрузчика достаточно просто дать команду RUN, и готовый программный продукт в виде трех файлов будет автоматически записан на внешний носитель. Теперь для запуска программы достаточно набрать LOAD "NAME".

Этот вариант при всех своих достоинствах, а именно простоте расчета адресов и минимальной суммарной длине файлов, не всегда удобен. Особенно это относится к компиляции больших программ. Действительно, когда верхняя граница области переменных скомпилированной программы подходит близко к нижней границе компилятора, есть смысл записать все это в виде одного файла, охватывающего область памяти от ADDR до ADDRCOMP+LENCOMP (см. табл. 4). Для таких случаев, главным образом, и предназначается следующий метод.

Метод второй

Он чрезвычайно прост, однако его применение сопряжено с несколько большим расходом памяти. На внешнем носителе формируются два файла: загрузчик на Бейсике и кодовый файл. Последний содержит скомпилированную программу, компилятор и некоторое количество неиспользованной памяти, находящейся между последним байтом скомпилированной программы и первым байтом области компилятора. Справедливости ради следует сказать, что эту область памяти нельзя считать полностью неиспользуемой, так как именно в ней располагается стек компилятора, но об этом чуть позже.

Для записи программного продукта этим методом необходимо составить следующий загрузчик на Бейсике:

```
10 CLEAR ADDR—1: REM Установка RAMTOP на адрес, на единицу
   меньший, чем начало скомпилированной программы
20 LOAD "NAME" CODE: REM Загрузка скомпилированной прог-
   раммы
30 RANDOMIZE USR ADDR: REM Запуск программы
```

Затем нужно выполнить строку операторов:

SAVE "NAME" LINE 10: SAVE "NAME" CODE ADDR, LENGTH

– где ADDR — адрес начала скомпилированной программы, LENGTH — длина всего блока в кодах, рассчитываемая по формуле:

LENGTH = ADDRCOMP + LENCOMP – ADDR

– где ADDRCOMP — адрес начала компилятора, а LENCOMP — его длина в байтах (см. табл. 4). Для простоты можно считать, что ADDRCOMP+LENCOMP=65535. Это позволит сохранить вместе с программой область определяемых пользователем символов, если, конечно, в этом есть необходимость. Естественно, что все операции по записи программы целесообразно, как и в первом варианте, поручить компьютеру. В этом случае программа-загрузчик будет иметь вид:

```
10 SAVE "NAME" LINE 40: REM Начало «самоспасателя»
20 SAVE "NAME" CODE ADDR, LENGTH
30 STOP: REM Окончание «самоспасателя»
40 CLEAR ADDR–1
50 LOAD "NAME" CODE
60 RANDOMIZE USR ADDR
```

И, наконец, для тех, кто, не считаясь с затратами времени, пожелает соединить преимущества двух описанных методов в одном, будет полезен третий метод.

Метод третий

Этот метод является наиболее трудоемким, но и самым эффективным. Его рекомендуется применять при создании законченного коммерческого продукта, так как именно этот способ удачно сочетает в себе как минимальное число файлов (два), так и наиболее эффективное использование места на магнитной ленте и объема памяти при работе программы. Основная идея этого метода заключается в размещении скомпилированной программы почти «впритык» к компилятору.

Метод требует специальной подготовки еще на этапе компиляции, поэтому приводим здесь подробный его алгоритм:

1. загрузить компилятор;
2. ввести исходный текст программы и запустить пробную компиляцию. По окончании запомнить стартовый адрес скомпилированной программы (ADDR) и адрес окончания области переменных (ENDVARS);
3. установить RAMTOP оператором

   CLEAR ADDRCOMP–(ENDVARS–ADDR)–WSPACE–1

   – где ADDRCOMP — адрес начала компилятора (см. табл. 4), ENDVARS и ADDR — соответственно, окончание и начало скомпилированной программы, запоминаемое после пробной компиляции, WSPACE — размер рабочего пространства (обыч-

но от 130 до 200 байт), резервируемого на нужды самого компилятора;

4. скомпилировать программу еще раз по новому адресу, после чего сохранить на внешнем носителе блок машинных кодов, выполнив оператор:

   SAVE "NAME" CODE ADDR, ADDRCOMP+LENCOMP–ADDR

   – где ADDRCOMP и LENCOMP — как и прежде, начальный адрес и длина компилятора (см. табл. 4).

Бейсик-загрузчик для считывания и автоматического запуска скомпилированных программ в этом случае будет выглядеть так:

```
10 CLEAR ADDR–1
20 LOAD "NAME" CODE
30 RANDOMIZE USR ADDR
```

Несмотря на свою эффективность, третий метод сохранения скомпилированных программ не исключает целесообразности применения остальных, которые наиболее удобны при необходимости экономии машинного времени, главным образом, на этапе разработки программного продукта.

Кроме того, при работе с дисководом первый метод (с раздельным сохранением компилятора и скомпилированной программы) позволяет экономить место на диске за счет сохранения единственного файла с пакетом рабочих процедур компилятора и нескольких файлов с результатами компиляции различных программ, запускаемых разными загрузчиками.

При необходимости можно обеспечить сохранение в одном файле нескольких программ, скомпилированных в разные адреса. В этом случае все скомпилированные программы записываются на внешний носитель в виде одного файла вместе с компилятором и после загрузки запускаются по разным адресам. Адреса нужно выбрать таким образом, чтобы скомпилированные программы размещались впритирку друг к другу: это позволяет минимизировать размер файла*. Для большего удобства и во избежание разрушения уже скомпилированных программ рекомендуется транслировать фрагменты, располагая их в памяти «сверху вниз». Для начала необходимо установить RAMTOP на максимально высокий адрес, позволяющий разместить над ним первый фрагмент и рабочую область компилятора. После этого следует скомпилировать первую программу, затем опустить RAMTOP на величину, необходимую для размещения следующей программы, и так до тех пор, пока не будут скомпилированы последовательно все программы. Затем остается лишь записать всю область памяти от RAMTOP до окончания компилятора на внешний носитель в виде единого блока кодов. Вызывающая программа пишется на Бейсике и также компилируется. Она содержит обращения к программам по адресам их начала. RAMTOP при загрузке устанавливается, как обычно, на адрес, на единицу меньший, чем адрес начала блока кодов.

---

* При осуществлении этого варианта предварительно может потребоваться отдельная компиляция программ с целью определения размера каждой из них.

Таким образом, появляется реальная возможность организации процесса программирования путем разделения глобальной задачи на несколько независимых модулей. Кроме того, можно создавать наборы предварительно скомпилированных системных библиотек, пригодных для использования в различных программах. Проблемой остается лишь способ обмена данными между отдельно скомпилированными модулями. Вследствие того, что для каждого программного фрагмента при компиляции формируется своя область переменных, для передачи значений переменных от одного фрагмента к другому требуются специальные приемы. Практически это можно осуществить, например, путем побайтового обмена данными с помощью операторов POKE и PEEK через специально отведенную область памяти.

Внешние процедуры в машинных кодах и таблицы пользователя (альтернативные символьные наборы и т. п.) могут располагаться в любом свободном месте памяти. Их можно оформлять как в виде отдельных файлов, так и присоединять к скомпилированной программе или самому компилятору. Впрочем, возможности человеческой фантазии поистине безграничны, поэтому предоставим читателю право самому придумать другие варианты использования богатейших возможностей, предоставляемых компиляторами Spectrum-Бейсика.

ПРИЛОЖЕНИЯ

1. Совместимость компиляторов и интерпретатора Бейсика

В предлагаемой таблице (табл. 5) представлены ограничения, налагаемые рассматриваемыми компиляторами на синтаксис операторов и функций Бейсика. Недопустимые инструкции отмечены знаком минус (–). Если инструкция допустима, но ее употребление имеет особенности по сравнению с интерпретатором, она отмечена цифрой, означающей номер примечания.

Таблица 5. **Ограничения синтаксиса при использовании компиляторов.**

Ключевые слова	ZX Compiler	MCoder 2	Softek IS	Softek FP 1.1	Softek FP 1.7	Tobos FP
Функции Бейсика						
ABS	+	+	+	+	+	+
ACS	–	–	–	+	+	+
AND	–	17	–	+	+	+
ASN	–	–	–	+	+	+

Ключевые слова	ZX Compiler	MCoder 2	Softek IS	Softek FP 1.1	Softek FP 1.7	Tobos FP
ATN	–	–	–	+	+	+
ATTR	+	+	+	+	+	+
BIN	+	+	+	+	+	+
CHR$	+	+	+	+	+	+
CODE	+	+	+	+	+	+
COS	–	–	–	+	+	+
EXP	–	–	–	+	+	+
FN	–	–	–	–	–	+
IN	+	+	+	+	+	+
INKEY$	1	+	+	+	+	+
INT	2	2	–	+	+	+
LEN	–	+	+	+	+	+
LN	–	–	–	+	+	+
NOT	–	–	+	+	+	+
OR	–	–	+	+	+	+
PEEK	+	+	+	+	+	+
PI	–	–	–	+	+	+
POINT	+	+	+	+	+	+
RND	3	3	3	+	+	+
SCREEN$	–	–	+	+	+	+
SGN	+	+	+	+	+	+
SIN	–	–	–	+	+	+
SQR	4	4	–	+	+	+
STR$	–	–	+	+	+	+
TAN	–	–	–	+	+	+
USR addr	+	+	+	+	+	+
USR "A"	5	5	5	+	+	+
VAL	–	–	–	6	+	+
VAL$	–	–	–	7	7	+

Ключевые слова	ZX Compiler	MCoder 2	Softek IS	Softek FP 1.1	Softek FP 1.7	Tobos FP
Операторы Бейсика						
AT	+	+	+	+	+	+
BEEP	18	18	+	+	+	+
BORDER	+	+	+	+	+	+
BRIGHT	+	+	+	+	+	+
CLEAR	+	8	8	+	+	8
CLOSE #	–	–	–	+	+	–
CLS	+	+	+	+	+	+
CONTINUE	–	–	–	–	–	–
COPY	+	+	+	+	+	+
DATA	+	+	+	+	+	+
DEF FN	–	–	–	–	–	+
DIM A()	–	9	–	9	9	+
DIM A$()	–	–	–	9	9	+
DRAW	+	+	+	+	+	+
FLASH	+	+	+	+	+	+
FOR...TO...STEP	10	10	+	+	+	+
GO SUB	11	11	11	11	11	+
GO TO	11	11	11	11	11	+
IF...THEN	+	+	+	+	+	+
INK	+	+	+	+	+	+
INPUT A	+	+	+	+	+	+
INPUT A$	–	+	+	+	+	+
INPUT LINE	–	–	–	6	7	+
INVERSE	+	+	+	+	+	+
LET A=	+	+	+	+	+	+
LET A$=	–	+	+	+	+	+
LIST, LLIST	–	–	–	–	–	12
LOAD	–	–	13	13	13	–

Ключевые слова	ZX Compiler	MCoder 2	Softek IS	Softek FP 1.1	Softek FP 1.7	Tobos FP
LPRINT	+	+	+	+	+	+
MERGE	–	–	–	–	–	–
NEW	+	+	+	+	+	+
NEXT	+	+	+	+	+	+
OPEN #	–	–	–	+	+	+
OUT	+	+	+	+	+	+
OVER	+	+	+	+	+	+
PAPER	+	+	+	+	+	+
PAUSE	+	+	+	+	+	+
PLOT	+	+	+	+	+	+
PRINT A	+	+	+	+	+	+
PRINT A$	–	+	+	+	+	+
PRINT #	–	–	+	+	+	+
RANDOMIZE	–	+	+	+	+	+
READ	+	+	+	+	+	+
REM	+	+	14	14	14	+
RESTORE	15	15	+	+	+	+
RETURN	+	+	+	+	+	+
RUN	–	–	–	–	–	+
SAVE	–	–	13	13	13	+
STOP	16	16	+	+	+	+
TAB	+	+	+	+	+	+
TO	–	–	+	+	+	+
VERIFY	–	–	13	13	13	–

Комментарии к таблице:

- При использовании целочисленных компиляторов любые дробные параметры в тексте программы диагностируются как ошибка и вызывают остановку компиляции.
- В среде ZX-Compiler v.1.0 не поддерживаются символьные переменные.

5 Зак. № 192

- Аргументы функций и параметры операторов в среде ZX Compiler v.1.0, представляющие собой выражения, должны заключаться в круглые скобки. Для MCoder 2 это ограничение распространяется только на операторы PLOT, DRAW и CIRCLE.
- Цифры в таблице обозначают:

1. функция поддерживается только в контексте CODE INKEY$;
2. избыточный оператор;
3. функция возвращает целочисленный результат;
4. дробная часть отбрасывается;
5. результат возвращается в виде числа со знаком в дополнительном коде (см. стр. 109);
6. инструкция компилируется, но приводит к сбою при выполнении программы;
7. компилируется, но работает неправильно;
8. допускается только без параметров;
9. массивы могут быть только одномерными;
10. инструкция STEP не поддерживается;
11. в качестве параметров нельзя использовать выражения и переменные;
12. выводится листинг программы, находящейся в области Бейсика, а не скомпилированной программы;
13. операторы работают только с файлами типа CODE;
14. может использоваться также для ввода операторов, расширяющих стандартный Бейсик;
15. допускается только с параметром;
16. воспринимается как ограничитель текста программы (аналогично END в ряде других языков высокого уровня);
17. может использоваться только в качестве условия в операторе IF...THEN;
18. в параметрах могут использоваться натуральные дроби.

2. Сравнительные характеристики компиляторов

В табл. 6 сведены данные о повышении скорости работы тест-программ, оттранслированных различными компиляторами, по сравнению со скоростью их выполнения интерпретатором Бейсика, а также размеры скомпилированных программ. Для проверки использовались 3 тест-программы: 1-я и 2-я целочисленные (экранная графика и целочисленная арифметика), а 3-я использует алгебраические операции с «плавающей точкой».

Тест 1

```
10 FOR A=0 TO 175
20 FOR B=0 TO 255
30 PLOT B,A
40 NEXT B: NEXT A
```

Тест 2

```
10 FOR A=1 TO 256
20 FOR B=1 TO 256
30 LET C=(A*B)-(A*B)
40 NEXT B: NEXT A
```

Тест 3

```
 10 FOR A=0 TO 255
 20 LET B=ABS (ACS (RND))+ASN (RND)-ATN (RND)
 30 LET B=ATTR (A,B)+BIN 10011110/CODE "q"
 40 LET A$=CHR$ A
 50 LET B=COS PI: LET B=SIN PI
 60 LET B=TAN PI-EXP PI: LET B=IN (31)-CODE INKEY$
 70 LET B=INT (2/3): LET B=LEN A$
 80 LET B=LN PI+PEEK 23681-POINT (128,88)
 90 LET A$=SCREEN$ (10,10)
100 LET B=SGN (RND)+SIN (RND)*SQR (ABS (RND))/TAN (RND)
110 LET A$=STR$ 12345
120 LET B=USR 7962+USR "a"
130 NEXT A
140 BEEP 0.1,0.1
```

Таблица 6. **Сравнение эффективности компиляторов.**

Тест	Бейсик	ZX Compiler	MCoder 2	Softek IS	Softek FP		TOBOS FP*	
					1.1	1.7	реж.1	реж.2
Степень ускорения (по сравнению с Spectrum-Бейсиком), крат								
1	1	16	16	35	3.7	3.7	6.2	6.6
2	1	17	28	18	3.1	3.1	7.0	7.5
3	1	—	—	—	1.1	1.1	19	19
Длина программы с переменными (без учета компилятора), байт								
1	108	115	134	142	92	109	145	126
2	145	154	171	189	166	166	171	148
3	379	—	—	—	409	409	416	354

* Режим 1 компиляции Tobos FP включается оператором **POKE 53252,0**; режим 2 – **POKE 53252,55** (подробнее см. стр. 119).

PRO-DOS

Система PRO-DOS v.1.1* расширяет возможности интерпретатора Spectrum-Бейсика, добавляя к стандартному списку операторов три десятка новых инструкций.

Перечислим основные «специальности» операторов PRO-DOS:

- вывод на экран графических объектов (треугольников, эллипсов, прямоугольников и т. д.);
- заливка контуров текстурой;
- выполнение операций с окнами;
- изменение размеров шрифтов;
- работа с теневым экраном.

Как видно, возможности PRO-DOS достаточно велики, особенно, если учесть, что в памяти компьютера он занимает всего 3787 байт (наименьший объем по сравнению с другими расширениями Spectrum-Бейсика, описанными в этой книге). Между кодами PRO-DOS и началом области символов, определяемых пользователем, существует зазор около 1,5 килобайт памяти (см. Приложение 2). Этот зазор недоступен бейсик-программам, но может быть использован для размещения программ в машинных кодах.

Отметим также, что PRO-DOS, в отличие, скажем, от MegaBasic и Beta Basic, не конфликтует с дисковой операционной системой TR-DOS: в программы можно смело включать инструкции обращения к дисководу.

Системный файл PRO-DOS имеет хождение вместе с демонстрационной программой, выполняющей по совместительству и функции загрузчика. Наличие виртуозно написанной демонстрационной программы, показывающей практически все возможности системы, избавляет нас от славословий в адрес PRO-DOS.

По нашему мнению, система PRO-DOS может быть очень полезна при составлении различных тестирующих и демонстрационных

---

* Автор Hans-Joachim Berndt, 1985 г. Файлы: PRODOSDEMO (тип BASIC, длина 7811 байт), PRO-DOS1.1 (тип CODE, длина 3787 байт, адрес загрузки 60000).

программ для образования, науки и бизнеса. Благодаря ее простоте работать с ней смогут даже начинающие программисты, тем более (мы надеемся) после прочтения этого описания. А для быстрейшего освоения приемов программирования в среде PRO-DOS рекомендуем впоследствии шаг за шагом изучить работу демонстрационной программы.

ЗАГРУЗКА И ЗАПУСК

Перед началом экспериментов (или работы) с PRO-DOS введите с клавиатуры строку

```
CLEAR 59999: LOAD "" CODE
```

и, «промотав» на ленте бейсик-файл демонстрационной программы, загрузите системный файл. Если демонстрационная программа уже загружена, очистите память с помощью оператора стандартного Бейсика NEW*. После этого необходимо выполнить следующую программку:

```
10 RANDOMIZE USR 60000
20 OPEN #2, "P"
```

Вообще, каждая программа, работающая с PRO-DOS, должна начинаться с этих строк. Строка 10 понятна: она запускает (инициализирует) PRO-DOS. А вот строка 20, связывающая канал "P" с потоком номер 2, может вызвать некоторое недоумение, поэтому кратко поясним ее назначение. Вывод информации на экран в стандартном Spectrum-Бейсике идет через канал "K"**. Однако через этот канал невозможно реализовать многие дополнительные возможности по выводу на экран, которые предоставляет система PRO-DOS. По каким-то соображениям автор PRO-DOS не стал задавать новый канал, а модифицировал канал "P" — единственный из стандартных каналов, поддающийся модификации***. Поэтому твердо усвоим, что программы, работающие с PRO-DOS, обязательно должны содержать строку 20. При ее отсутствии будут утрачены наиболее ценные возможности PRO-DOS: доступность нижней части экрана для вывода графики, возможность изменения размеров шрифтов и др.

Во все имеющиеся в описании PRO-DOS примеры программ необходимо добавлять две приведенные строки с номерами 10 и 20 (в целях упрощения записи они опущены). Для запуска примеров программ используйте команды RUN или GO TO 10.

---

* Обратите внимание, что демонстрационная программа производит модификацию некоторых символов, определяемых пользователем (UDG).

** Подробно о потоках и каналах смотрите в [1].

*** При этом, естественно, возникли сложности с выводом на принтер, так как канал "P" еще Клайвом Синклером предназначался именно для этих целей. Как разрешить проблему вывода на принтер из PRO-DOS, объяснено в Приложении 3.

КЛЮЧЕВЫЕ СЛОВА

Ключевые слова PRO-DOS вводятся по буквам. Для того чтобы компьютер не принял их за имена переменных, перед ними ставится префикс — «звездочка» (*). Не надо пытаться «впихивать» звездочку перед ключевыми словами стандартного Бейсика, например, вводить *PLOT, сначала нажав клавишу Q (PLOT), а затем звездочку — PRO-DOS этого не поймет.

Операторы PRO-DOS можно вводить как прописными, так и строчными буквами. Пробелы между ключевым словом и параметрами оператора необязательны*. Из всего сказанного следует, что приведенные ниже инструкции PRO-DOS воспримет как идентичные:

```
*WINDOW C, a1, q(14), VAL A$
*window c, A1, Q(14), VAL a$
```

и даже

```
*wInDoWc,a1,q(14),VAL A$
```

В тексте для облегчения восприятия мы будем записывать операторы исключительно прописными буквами и с пробелами.

По написанию некоторые операторы PRO-DOS отличаются от операторов Spectrum-Бейсика лишь наличием префикса-звездочки. Но это еще не свидетельствует об их функциональной идентичности. Такие операторы могут быть и очень близки по смыслу (как *PLOT и PLOT), и иметь лишь некоторое сходство (*CLS и CLS), а также весьма отдаленно напоминать друг друга (*NEW и NEW) и, наконец, не иметь между собой ничего общего (*LINE и LINE).

Иногда работа стандартных бейсиковских операторов (таких как PRINT, AT, TAB, а также операторов установки атрибутов) в PRO-DOS-программах несколько отличается от того, что мы привыкли от них ожидать. Такие случаи, по возможности, будут рассмотрены.

Краткая справочная информация об операторах PRO-DOS приводится в Приложении 1.

Но достаточно вступительных слов, теперь — за дело!



---

* Наличие пробелов улучшает читаемость текстов программ, а отсутствие — экономит память.

ГРАФИЧЕСКИЕ ОПЕРАТОРЫ

*PLOT, *DRAW, *LINE, *BOX, *TRIANGLE, *ELLIPSE, *GPAT

Стандартный Spectrum-Бейсик предоставляет в распоряжение пользователя три графических оператора: PLOT, DRAW и CIRCLE. Диалект PRO-DOS дополняет этот список еще семью.

Операторы

***PLOT x, y**
***DRAW x, y**

очень похожи на свои стандартные аналоги. Отличие заключается в том, что *PLOT и *DRAW доступна вся площадь экрана, включая две нижние символьные строки (служебный экран). Следовательно, для этих операторов размеры экрана составляют не 256×176 точек, как для операторов стандартного Бейсика, а 256×192. Нелишне будет заметить, что для функции стандартного Бейсика POINT (x, y) (см. стр. 88), информирующей о состоянии пикселя (включен-выключен), служебный экран по-прежнему остается недоступным.

Поскольку начало отсчета координат привязано к нижнему левому углу экрана, то для графических операторов PRO-DOS оно смещается на 16 пикселей вниз. Следовательно, результаты выполнения операторов DRAW и PLOT, с одной стороны, и *DRAW и *PLOT, с другой стороны, при одинаковых параметрах не будут совпадать. В этом легко убедиться, выполнив следующую программу:

```
30 PLOT 30,30: DRAW 70,70: REM Стандартные операторы
40 *PLOT 30,30: *DRAW 70,70: REM Операторы PRO-DOS
50 PAUSE 0
```

Для полноты картины следует еще добавить, что *DRAW, в отличие от DRAW, не умеет рисовать дуги и вообще не «переваривает» введение трех параметров.

Человек, мало-мальски знакомый с иностранными языками, легко сообразит, что операторы *LINE, *BOX, *TRIANGLE, и *ELLIPSE строят на экране, соответственно, отрезок прямой, прямоугольник, треугольник и эллипс. Этим операторам также доступны все 24 символьные строки экрана. Формат первых двух операторов одинаков:

***LINE x1, y1, x2, y2**
***BOX x1, y1, x2, y2**

В первом операторе двумя парами значений задаются координаты концов отрезка, во втором — координаты расположенных по диагонали вершин прямоугольника. Так, программа

```
30 LET x1=20: LET x2=120: LET y1=50: LET y2=150
40 *LINE x1, y1, x2, y2
50 *BOX x1, y1, x2, y2
60 PAUSE 0
```

нарисует на экране прямоугольник и его диагональ. Отметим, что инструкции

*LINE x1, y1, x2, y2

и

*LINE x2, y2, x1, y1

эквивалентны. То же самое касается и различных вариантов записи оператора *BOX:

*BOX x1, y1, x2, y2
*BOX x2, y2, x1, y1
*BOX x1, y2, x2, y1

и

*BOX x2, y1, x1, y2

К сожалению, стороны прямоугольника, построенного оператором *BOX, всегда параллельны осям координат.

Оператор *TRIANGLE требует 6 параметров:

***TRIANGLE x1, y1, x2, y2, x3, y3**

В списке параметров фигурируют пары «абсцисса-ордината» трех вершин треугольника, причем порядок перечисления этих пар непринципиален.

Если операторы *LINE, *BOX и *TRIANGLE являются по сути избыточными (их действие легко смоделировать с помощью *PLOT и *DRAW), то оператор *ELLIPSE уникален: ничего похожего нет ни в стандартном Spectrum-Бейсике, ни в каком-либо из известных его диалектов. Он строит эллипс, что сделать с помощью других операторов не так-то просто (хотя и возможно). Следом за ключевым словом *ELLIPSE указываются 4 параметра:

***ELLIPSE x, y, a, b**

Первые два параметра (x, y) определяют координаты центра эллипса, вторые (a, b) — соответственно, горизонтальную и вертикальную полуоси. И опять же, как и стороны прямоугольника, выводимого оператором *BOX, полуоси эллипсов всегда параллельны границам экрана.

Окружность, как известно, является частным случаем эллипса. Для ее построения нужно задать одинаковые длины полуосей. Действительно, оператор

*ELLIPSE x, y, r, r

изобразит на экране окружность радиуса r с центром в точке с координатами (x, y). Такая альтернатива оператору CIRCLE может пригодиться при построении окружности в области служебного экрана.

Для ценителей гармонии линий приведем пример программы, рисующей семейство концентрических эллипсов:

```
30 LET x=128: LET y=96: REM Центр эллипса
40 FOR a=10 TO 100 STEP 20: REM Горизонтальная полуось
50 FOR b=10 TO 100 STEP 20: REM Вертикальная полуось
60 *ELLIPSE x, y, a, b
70 NEXT b: NEXT a: PAUSE 0
```

Несколько замечаний, общих для всех графических операторов PRO-DOS. В отличие от стандартных, они не допускают использования временных атрибутов. Так, компьютер откажется «глотать» конструкцию вроде

*PLOT INK 5; 30, 40

Более того, графические операторы совершенно равнодушны к постоянным атрибутам, установленным инструкциями INK, PAPER и другими (не в составе оператора PRINT).

Цветную графику можно получить, лишь вставив в программу фиктивный оператор печати: ничего не печатающий, но устанавливающий атрибуты, например:

PRINT INK 5; OVER 1; BRIGHT 0

Попробуйте модифицировать программу построения эллипсов, добавив в нее под номером 35 приведенную строку.

Графические операторы PRO-DOS иначе, чем их стандартные аналоги, реагируют на выход за границы экрана в процессе построения линий. Предоставим читателю возможность самостоятельно поэкспериментировать, присваивая параметрам графических операторов большие значения (но не более 255). При этом особенно забавные эффекты будут получаться при выполнении оператора *ELLIPSE. В отличие от стандартных графических операторов, при выходе линии за пределы экрана в ряде случаев не выдается сообщение об ошибке, например

*PLOT 128,96: *DRAW 200, 200

или

*ELLIPSE 128,96,50,100: PAUSE 0

Действие всех графических операторов PRO-DOS может быть модифицировано с помощью оператора

***GPAT n**

Он устанавливает вид линий, которыми будут производиться графические построения: сплошные, с разрывами или вообще невидимые.

*GPAT относится к *неисполняемым* операторам (подобно INK, PAPER и другим). Это означает, что сам он не вызывает никаких действий, а лишь влияет на результат выполнения некоторых последующих инструкций. Позволим себе привести в качестве аналогии уставные армейские команды «ШАГОМ» и «БЕГОМ», которые сами не оказывают никакого видимого воздействия на дисциплинированных военнослужащих, а лишь сообщают, что им придется делать после команды «МАРШ». Операторы такого типа характерны для PRO-DOS, и мы еще не раз с ними встретимся.

Вид линии для графических построений задается параметром n (0...255), следующим за оператором *GPAT. Его значение, записанное в двоичном виде, наглядно представляет периодически повторяющийся элемент линии — 8 пикселей, из которых включенным пикселям соответствуют единицы, а погашенным — нули. Удобно

вводить параметр в операторе *GPAT в двоичном виде с помощью функции BIN. Например, вместо

```
35 *GPAT 105
```

записывать

```
35 *GPAT BIN 01101001
```

Попробуйте вставить приведенную строку в пример программы, рисующей эллипсы (рис. 7).

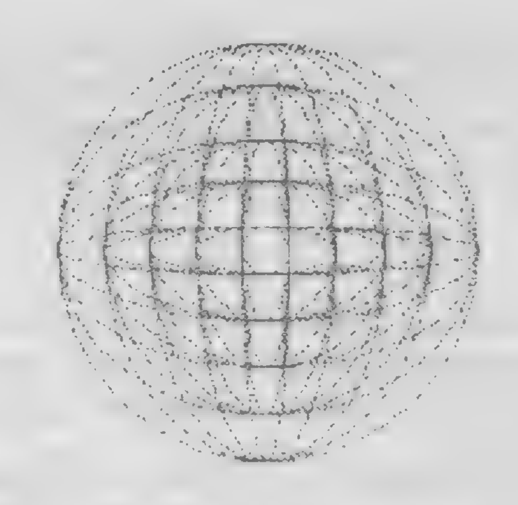


Рис. 7. Построения прерывистыми линиями.

Выполнение *GPAT с нулевым аргументом заставляет графические операторы «строить» фигуры невидимыми линиями. Оператор *GPAT 255 восстанавливает режим рисования сплошной линией; этот же режим устанавливается при инициализации PRO-DOS.

*GPAT имеет и еще одно полезное применение, о котором логичнее рассказать в следующем разделе.

ЗАКРАШИВАНИЕ КОНТУРОВ

*FBOX, *FILL, *PAINT, *HATCH, *MATCH

Мы выяснили, как рисовать на экране контуры геометрических фигур. Теперь неплохо бы научиться их закрашивать.

Единственной фигурой, которую можно получить на экране сразу закрашенной (или, как говорят, *залитой*) без предварительного прочерчивания контура, является прямоугольник. Его построение и заливку делает оператор

***FBOX x1, y1, x2, y2**

Формат этого оператора совпадает с форматом оператора *BOX. Фигуры могут быть залиты не только сплошным фоном, но и неким периодически повторяющимся рисунком — текстурой. При работе с *FBOX текстура задается оператором *GPAT, рассмотренным в предыдущем разделе. Для примера выполним программу:

```
30 *GPAT BIN 10000000: REM Шаблон линии
40 *FBOX 30, 30, 130, 100: REM Прямоугольник
50 PAUSE 0
```

Текстура для закраски прямоугольника формируется из вплотную прилегающих друг к другу горизонтальных линий. Линии составляются из отрезков длиной в 8 пикселей (период трансляции равен 8), структура которых задается оператором *GPAT. Если в пределах ширины прямоугольника помещается дробное число таких отрезков, то следующая линия будет начинаться с того места отрезка, где закончилась предыдущая. Это приводит к тому, что при неизменном параметре оператора *GPAT (то есть при одинаковом виде линий) будут получаться различные текстуры для прямоугольников с различными остатками от деления их ширины на

число 8. Линии получаются по-разному сдвинутыми относительно друг друга. Последнее станет яснее, если выполнить несколько раз приведенный выше пример, заменяя в 40-й строке число 130 на 131, 132 и т. д. вплоть до 138.

*FBOX — единственный оператор заливки, работающий столь необычным образом. Ко всем прочим операторам этого раздела *GPAT никакого отношения не имеет.

Если возникает необходимость закрасить контур произвольной формы, его надо сначала построить. Для этого можно воспользоваться как средствами стандартного Бейсика, так и операторами PRO-DOS. Если применяются последние, то нужно следить, чтобы они «рисовали» обязательно сплошными линиями, иначе заливка «просочится» сквозь контур. Во избежание неприятностей перед экспериментами с заливкой лучше установить режим рисования сплошной линией, введя прямо с клавиатуры *GPAT 255.

***FILL x, y**

выполняет сплошную (не текстурную) заливку. Параметрами x и y задается координата любой точки внутри закрашиваемого контура. Попытка указать координаты точки, расположенной за пределами замкнутого контура, приведет к сообщению об ошибке (это замечание относится ко всем операторам заливки). Продемонстрируем работу оператора *FILL следующей программой:

```
30 *BOX 30, 70, 140, 100: REM Прямоугольник
40 *ELLIPSE 50, 50, 45, 30: REM Перекрывающий его эллипс
50 *FILL 80, 80: REM Сплошная заливка
60 PAUSE 0
```

Простейшим из операторов, позволяющих выполнять текстурную заливку контура, является

***PAINT x, y**

Формат его совпадает с форматом оператора *FILL. Заменим в последнем примере *FILL на *PAINT, оставив все прочее без изменений. Когда программа отработает, мы увидим ровный, но не слишком оригинальный узор (рис. 8). Получить другой узор с помощью *PAINT, увы, невозможно.

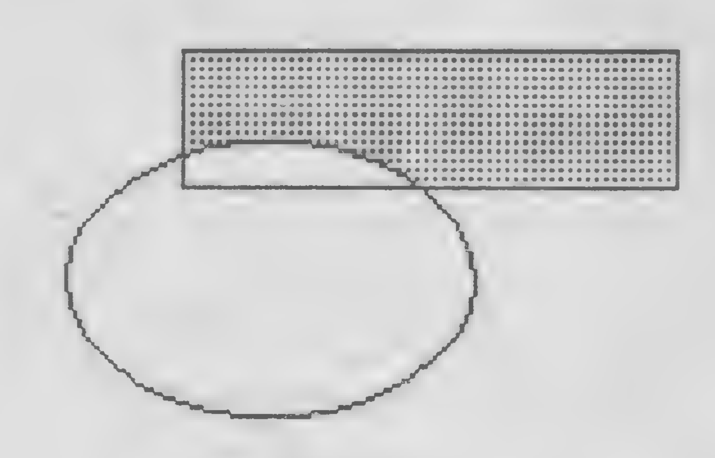
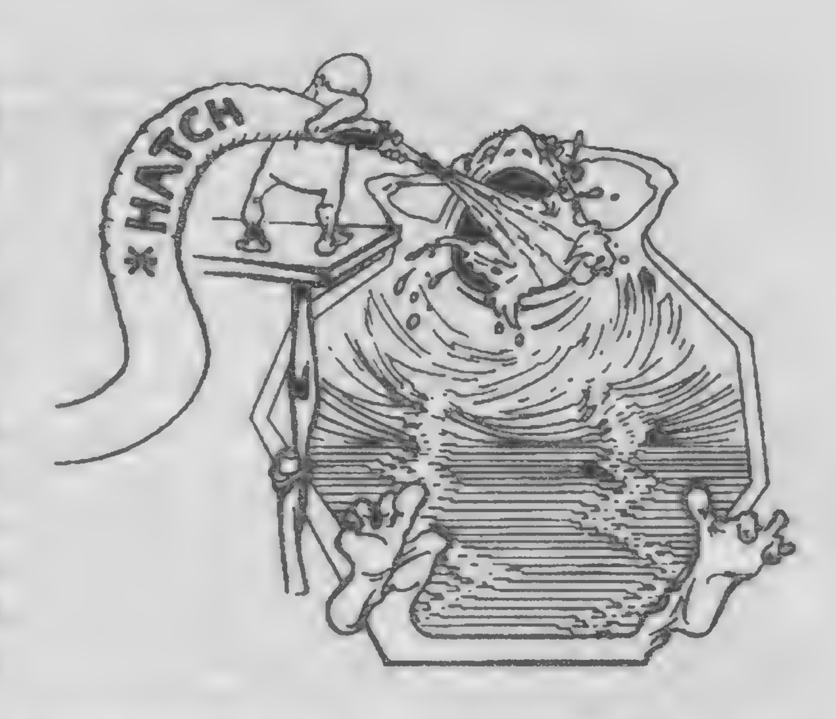


Рис. 8. Заливка оператором *PAINT.

Более универсальными в группе операторов, выполняющих заливку, являются *HATCH и *MATCH.

***HATCH x, y, addr**

позволяет заливать замкнутые контуры любым узором с периодом трансляции 8 пикселей по вертикали и 8 по горизонтали.



Первые два параметра (x, y) задают координаты любой точки внутри контура (как в *FILL и *PAINT). Третий (addr) — адрес (0...65535) начала 8-байтовой области памяти, где закодировано изображение некоего символа. Этим символом и будут заполнены все знакоместа (8×8 пикселей) внутри контура. В знакоместах, пересеченных контуром, прорисовывается лишь часть символа, попавшая вовнутрь контура.

Изображение символа формируется стандартным для ZX Spectrum способом, поэтому создавать текстуру для заливки Вы можете сами. Для этих целей удобно пользоваться областью символов, определяемых пользователем (UDG), где после включения компьютера размещаются изображения прописных латинских букв от A до U. Выполним программу:

```
30 *BOX 30, 70, 140, 100
40 *ELLIPSE 50, 50, 45, 30
50 *HATCH 80, 80, USR "P"
60 PAUSE 0
```

Контур будет заполнен буквами P, изображение которых хранится в 8 ячейках памяти, начиная с адреса 65488 (значение, возвращаемое функцией USR "P"). Изменяя эти 8 байт, можно создать любую текстуру. Оригинальные узоры получаются, если подставлять в оператор *HATCH произвольные адреса памяти ZX Spectrum (например, адреса в ПЗУ). Замените в примере USR "P" на 0, 1, 2 и т. д. и посмотрите, что получится.

Интересно также взглянуть, что произойдет, если воспользоваться оператором *HATCH там, где он один раз уже отработал, то есть попытаться «перекрасить» залитый контур:

Рис. 9. «Перекраска» текстуры оператором *HATCH.

```
30 *BOX 30, 30, 140, 100
40 *ELLIPSE 50, 50, 45, 30
50 *HATCH 80, 80, USR "P"
60 *HATCH 80, 80, USR "T": REM Заменится ли текстура?
70 PAUSE 0
```

Не правда ли, это не совсем то, что мы ожидали (рис. 9)? При повторной заливке текстуры наложились одна на другую необычным образом.

Полностью заменить текстуру (то есть предварительно стереть предыдущую) можно, выполнив оператор

***MATCH addr**

Его действие относится к контуру, который последним был закрашен с помощью оператора *HATCH (но не *PAINT и не *FILL). Проверим это, заменив в предыдущем примере строку 60 на

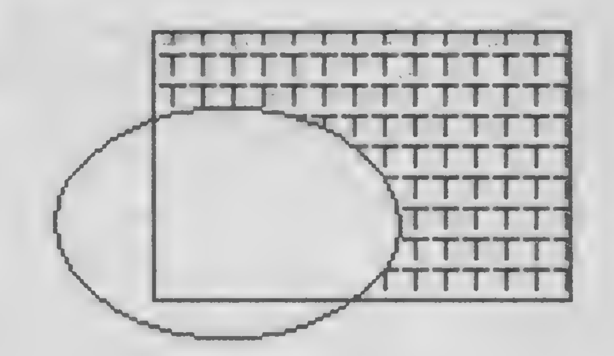


Рис. 10. «Перекраска» текстуры оператором *MATCH.

```
60 *MATCH USR "T": REM Теперь точно зальется!
```

Вот теперь, действительно, произошла смена текстуры (рис. 10).

ОКНА

*WINDOW, *WSIZE, *NEW, *WRAP, *NOWRAP, *ROLL, *SCROLL, *CLEAR, *CLS, *TPAT

В PRO-DOS имеется группа операторов, предназначенных для работы с *текстовыми окнами*. Текстовым окном называется прямоугольная область экрана, в которую может быть направлен вывод символьной информации. В PRO-DOS размеры окна должны составлять целое число знакомест.

До того как выполнен первый оператор работы с окнами, считается, что в PRO-DOS определено восемь одинаковых окон размером во весь экран (32×24 знакоместа). Если такое равноправие Вас не устраивает, прежде всего укажите, с каким окном Вы намерены работать, то есть определите *текущее окно*. Делается это с помощью оператора

***WINDOW n**

Параметр n задает номер окна и может быть в пределах от 0 до 255. Однако учитываются лишь три младших бита этого числа, по которым и определяется номер окна (с 0 по 7). Следовательно, подставив вместо n число 8, мы назначим текущим нулевое окно, 9 — первое и т. д. Оператор *WINDOW n, как и *GPAT, является неисполняемым. Он лишь указывает последующим операторам, в какое окно осуществлять вывод.

Оператор *WSIZE служит для установки размеров текущего окна. Его формат:

***WSIZE X1, Y1, X2, Y2**

Параметрами X1 и Y1 задаются координаты левого верхнего угла окна, X2 и Y2 — правого нижнего. Мы намеренно обозначили координаты не строчными, как обычно, а прописными латинскими буквами, чтобы показать, что система «оконных» координат отличается от той, в которой работают графические операторы. Вообще говоря, это та же система координат, которая применяется

в стандартном Бейсике для обозначения позиций печати, то есть все величины измеряются в знакоместах, а началом отсчета (0, 0) считается верхнее левое знакоместо экрана.

Оператор

***NEW**

деинициализирует все ранее установленные окна, и опять каждое из 8 окон будет занимать целиком весь экран.

После того, как окно произвольного размера устанавливается текущим (оператор *WINDOW), вывод информации в него будет осуществляться так, как будто ничего не изменилось, только экран стал меньше. Правда, при заполнении окна Вы будете избавлены от характерного для стандартного Бейсика вопроса scroll?. Но скроллинг все же происходить будет, причем помимо Вашего желания. Давайте поэкспериментируем:

```
30 *WINDOW 0: REM Работаем с окном номер 0
40 *WSIZE 10, 10, 20, 20: REM Установим его размер
50 PRINT "Window No. 0": REM Где окажется надпись?
60 PRINT AT 3,3; "AT 3,3": REM А эта?
70 PRINT "a string" ' "next string": REM Как работает символ апост-
   рофа?
80 PAUSE 200: INK 6: PAPER 7: REM А атрибуты?
90 LIST: REM Неужели листинг тоже выведется в окно?
```

Из результата выполнения строки 60 видно, что позиция печати отсчитывается от левого верхнего угла окна, а не всего экрана.

Обратите внимание, что работа операторов стандартного Бейсика PRINT и LIST в PRO-DOS имеет некоторые особенности. Как уже отмечалось, введение в программу строки

```
OPEN #2, "P"
```

сделает доступными для вывода две строки (22 и 23), относящиеся к служебному экрану. Встретив эту инструкцию, Speccy уже не станет возражать против команд типа PRINT AT 22,n и PRINT AT 23,n и честно выполнит их, если они имеют смысл для текущего окна. Оператор вывода текста в строку, номер которой больше высоты окна (вплоть до печати в строку номер 255), приведет к скроллингу на соответствующее число строк и присвоит знакоместам новых строк текущие атрибуты. Для лучшего понимания сказанного полезно посмотреть, как работает следующая программа:

```
30 FOR N=0 TO 30: REM Попробуем вывести 31 строку
40 PRINT AT N,0; INK INT(RND*7+.5); PAPER INT(RND*7+.5); FLASH
   INT(RND+.5); N: REM Да еще в цвете
50 PAUSE 100: NEXT N: REM И не спеша
```

Здесь в качестве окна используется весь экран. Интересно, что PRO-DOS «пропускает» и инструкции типа PRINT AT n,m, где m больше ширины текущего окна (вплоть до 255). Такие операторы просто игнорируются. Добавив в конец предыдущего примера строку

```
70 PRINT AT 0, 200; "***"
```

можно в этом убедиться.

Большинство управляющих символов, используемых в операторе PRINT, работает как обычно, кроме запятой, которая может и игнорироваться.

В некоторых случаях описанные особенности операторов вывода на экран являются нежелательными. Например, может вызвать трудности отсутствие запроса scroll? при просмотре листинга программы или выводе на экран каких-нибудь длинных таблиц. Эти трудности легко преодолимы. Самый простой способ вернуть операторам PRINT и LIST их исконные, принятые в стандартном Бейсике свойства — отключить на время канал "P", например:

```
CLOSE #2: LIST: OPEN #2,"P"
```

В этом случае листинг будет выдаваться с запросом scroll? и не в текущее окно, а как полагается, на весь экран.

Если в программе требуется выводить данные на экран как «продосовским», так и стандартным способами, проще для последнего открыть специальный поток. Сделать это можно, включив в программу строку

```
30 OPEN #4, "S"
```

После этой строки операторы PRINT и LIST будут работать так, как им положено это делать в PRO-DOS, а стандартный бейсиковский вывод будут осуществлять операторы PRINT #4 и LIST #4.

Скроллинг в окне можно заблокировать оператором

***WRAP**

После его выполнения текст будет выводиться с верхней левой позиции окна, замещая уже существующий текст. Впрочем, опять же без какого-либо запроса на разрешение сделать это. Вернуться к обычному скроллингу можно с помощью оператора

***NOWRAP**

После инициализации PRO-DOS всегда по умолчанию устанавливается режим *NOWRAP.

Для принудительного скроллинга изображения в окне служат два похожих оператора: *ROLL и *SCROLL. Оба имеют одинаковый формат:

***SCROLL m, n**
***ROLL m, n**

Параметром n определяется направление скроллинга. Из соображений наглядности разработчик PRO-DOS связал значения этого параметра с цифровыми обозначениями соответствующих курсорных клавиш ZX Spectrum:

n=5	влево;
n=6	вниз;
n=7	вверх;
n=8	вправо;
n=5+6=11	влево-вниз;
n=5+7=12	влево-вверх;
n=8+6=14	вправо-вниз;
n=8+7=15	вправо-вверх.

Вообще говоря, параметр n может принимать любые значения в диапазоне −255...255, однако предсказуемый эффект дадут лишь приведенные выше.

Параметром m (−255...255) задается количество знакомест, на которое нужно осуществить смещение текста в окне (в направлении, указанном параметром n). Параметр m, равный нулю, заблокирует скроллинг.

В чем же отличие между операторами *SCROLL и *ROLL? Первый осуществляет обычный (нециклический) скроллинг. После его выполнения текст при смещении будет уходить «в никуда». *ROLL реализует циклический скроллинг: символы, исчезнувшие у одной границы окна, тут же будут появляться у противоположной.

*CLEAR n

Оператор *CLEAR дает возможность быстро очищать окно и устанавливать в нем цветовые атрибуты. Значения параметра n могут быть любыми в диапазоне 0...255, хотя интерес представляют лишь первые четыре числа (дальше все повторяется). Оператор *CLEAR 3 очистит окно и установит для всех его знакомест значения атрибутов, которые последними встретились при выполнении программы. Причем неважно, какие это были атрибуты: постоянные или временные. Так, приведенная ниже программа, в конечном итоге, очистит указанное окно и окрасит его в красный цвет (PAPER 2) без мерцания (FLASH 0):

```
30 *WINDOW 0: REM Текущее окно
40 *WSIZE 3, 3, 10, 10: REM Размеры окна
50 FLASH 1: PRINT INK 3; PAPER 4; "***"; INK 1; PAPER 2; "+++";
   FLASH 0;
60 *CLEAR 3: REM Очистка окна
```

*CLEAR 2 сделает все то же самое, но не удалит ранее выведенные в окно символы. *CLEAR 1, напротив, очистит окно от символов, а атрибуты оставит без изменений. Вы можете убедиться в этом, меняя параметр оператора *CLEAR в строке 60 приведенной программы. Таким образом, результат последовательного выполнения *CLEAR 1 и *CLEAR 2 эквивалентен тому, что получится при выполнении *CLEAR 3. Что касается *CLEAR 0, то он вообще не вызывает никаких изменений.

Но возможности оператора CLEAR шире. Выполните пример:

```
30 *WINDOW 0
40 *WSIZE 3, 3, 10, 10
50 PRINT INK 3; PAPER 4: REM Пока все понятно...
60 *TPAT BIN 01001100: REM ... а это что-то новенькое
70 *CLEAR 3
```

и обнаружите, что окно залито текстурой в соответствии с шаблоном, заданным оператором

*TPAT

Попробуйте изменить параметр оператора *CLEAR в строке 70. Вы увидите, что ноль снова не приведет ни к какому эффекту, единица даст черно-белую заливку окна текстурой, а двойка:

закрасит окно атрибутами (в соответствии со строкой 50), но без текстурной заливки.

Оператор

***CLS**

делает в точности то же самое, что и *CLEAR 3, и введен, по-видимому, для удобства. Совместная работа *TPAT и *CLS напоминает действие пары *GPAT и *FBOX. Однако для *TPAT и *CLS соседние горизонтальные линии заливки смещаются друг относительно друга на один пиксель, и это приводит к тому, что узор в окне имеет вид диагональных полосок. По-видимому, не требует особых пояснений утверждение, что

```
*TPAT 255: *CLS
```

даст сплошную заливку окна, а

```
*TPAT 0: *CLS
```

очистит окно от текстуры. Повторное использование этих двух операторов в одном и том же окне приведет к замене текстуры на новую.

Напоследок отметим, что графические операторы при выводе на экран игнорируют факт существования окон.

Некоторые дополнительные сведения об окнах приведены в Приложении 4.

ШРИФТЫ

*CCHR, *CHR, *CSIZE, *LARGE, *NORMAL, *DIR

Подобно таким мощным диалектам Бейсика, как Mega-Basic и старшие версии Beta Basic, маленький PRO-DOS позволяет работать со шрифтами различных размеров.

Простейшей операцией со шрифтом, выполняемой PRO-DOS, является переключение набора символов со стандартного (ширина знака 8 пикселей) на узкий (ширина знака 4 пикселя). Делает это переключение оператор

***CCHR**

Применяется узкий шрифт так же, как и обычный (с учетом приведенных в предыдущем разделе особенностей вывода данных в окна). Только атрибуты для двух смежных символов, вписанных в стандартное знакоместо (8×8 пикселей), естественно, будут всегда одинаковы. Кроме того, узкие символы не распознаются функцией стандартного Бейсика SCREEN$ (X,Y) (см. стр. 92).

После переключения на узкий шрифт произойдет переопределение всех зарегистрированных в системе окон. За единицу измерения величин, определяющих расположение и размеры окна, будет принято не стандартное, а узкое знакоместо (4×8 пикселей).

Приведем пример, который иллюстрирует эту достаточно неординарную особенность:

```
 30 *WSIZE 10, 2, 16, 8: REM Единственное окно
 40 *TPAT BIN 11100010: REM Шаблон для его заливки
 50 GO SUB 100: REM Сходим на заливку
 60 *CCHR: REM Узкий шрифт
 70 PAUSE 100: GO SUB 100: REM Зальемся еще разок
 80 STOP
100 *CLS: PRINT "***": RETURN: REM Здесь заливают
```

Результат работы программы приведен на рис. 11.

Создается впечатление, что на экране присутствуют два различных окна. На самом же деле, это одно и то же, определенное в строке номер 30 окно, которое по-разному трактуется в зависимости от типа установленного шрифта. Попутно заметим, что изменение размера шрифта не влияет на фактуру заливки, заданную инструкцией *TPAT.

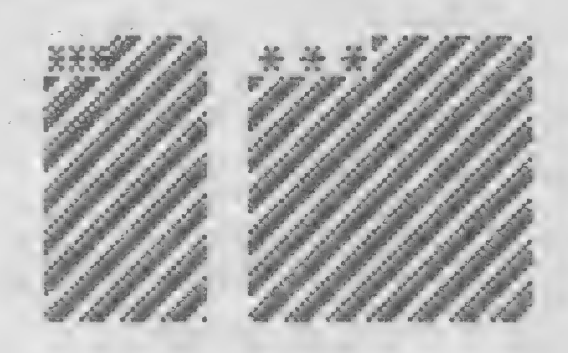


Рис. 11. «Раздвоение» окон.

Если узкий шрифт инициализирован сразу после запуска PRO-DOS или после команды *NEW, то для него будет доступна лишь левая половина экрана. Строка, длиннее 32 символов, будет переноситься, а попытка выполнить, скажем, оператор

```
PRINT AT 0, 40; "***"
```

будет игнорироваться. Чтобы сделать доступной для узкого шрифта правую половину экрана, необходимо создать окно соответствующего размера, например, оператором

```
*WSIZE 0, 0, 63, 23
```

Восстановить режим печати нормальным шрифтом, то есть отменить действие *CCHR, можно при помощи оператора

***CHR**

Если узкий шрифт по своим свойствам практически аналогичен обычному, то шрифты, которые мы с известной долей условности будем называть *графическими*, существенно отличаются от него.

Размер графических шрифтов задается неисполняемым оператором

***CSIZE n, m**

и может быть только кратным (и, в частности, равным) размеру стандартного. Первый параметр определяет значение вертикального, а второй — горизонтального масштабных факторов. Оба значения могут быть только целочисленными. Например, инструкция *CSIZE 2, 1 «сгенерирует» шрифт в два раза выше, а *CSIZE 1, 2 — в два раза шире стандартного. Команда *CSIZE 32, 24 сформирует шрифт, где каждый символ будет размером во весь экран. Возможно и дальнейшее увеличение размеров символа (вплоть до

*CSIZE 255, 255), равно как и уменьшение до нулевого размера (*CSIZE 0, m, *CSIZE n, 0 или *CSIZE 0, 0). Однако эти варианты вряд ли представляют практический интерес.

Поскольку оператор *CSIZE является неисполняемым, в PRO-DOS введен специальный оператор инициализации графического шрифта

***LARGE**

Он не требует параметров. После того как *LARGE встретится в программе, PRINT начнет печатать графическим шрифтом, размеры которого были заданы последним выполненным оператором *CSIZE. Обратное переключение на стандартный шрифт производит оператор

***NORMAL**

*LARGE без предварительного выполнения оператора *CSIZE устанавливает графический шрифт с размером, равным стандартному (8×8 пикселей). То есть по умолчанию считается, что выполнен оператор *CSIZE 1,1.

Теперь о специфике работы с графическими шрифтами. Во-первых, при печати с их использованием игнорируются ключевые слова AT и TAB — оператор PRINT выводит символы, начиная с левого верхнего угла экрана. Если текст превышает длину строки экрана, то вывод продолжается с начала той же строки.

Позиция вывода устанавливается с помощью оператора *PLOT x, y (но не PLOT без звездочки!). Параметры x (0...256) и y (0...192) задают координаты верхнего левого угла позиции печати первого выводимого символа. Для того, чтобы в этой позиции не появлялась точка, можно использовать конструкцию

```
PRINT INVERSE 1;: *PLOT x, y: PRINT INVERSE 0
```

При выводе на экран надписей, выполненных графическими шрифтами, так же как и при графических построениях, игнорируются знакоместа и окна, символы накладываются на существующее изображение, не стирая его. Это и позволило нам называть их «графическими».

При работе с графическими шрифтами нужно следить за цветовыми атрибутами экрана. Может получиться так, что один символ будет окрашен в несколько цветов (хотя это не всегда плохо).

Уникальной среди диалектов Бейсика является способность PRO-DOS печатать графическими шрифтами в восьми направлениях. Неисполняемый оператор

***DIR n**

определяет направление печати в зависимости от значения параметра n, которое задается тем же способом, что и для операторов скроллинга (см. стр. 143).

Приведем пример, демонстрирующий возможности графических шрифтов (рис. 12):

```
30 *NORMAL: REM Стандартный шрифт
40 FOR L=2 TO 15: PRINT AT L, 6; "NORMAL": NEXT L:REM Строки
    стандартным шрифтом
```

```
50 *CSIZE 2, 2: REM Размер графического шрифта
60 *LARGE: REM Инициализация графического шрифта
70 *DIR 14: REM Пишем вправо-вниз
80 PRINT INVERSE 1;: *PLOT 20, 180: PRINT INVERSE 0;
90 PRINT "GRAPHIC": REM Строка графическим шрифтом
```

Символы графических шрифтов различных размеров формируются непосредственно в процессе работы программы из текущего набора символов, расположенного в ПЗУ или ОЗУ. Поэтому при использовании альтернативного (например, русифицированного) символьного набора автоматически будут получаться графические шрифты такого же вида. Напротив, изображения символов узкого шрифта закодированы внутри системного файла PRO-DOS, и поэтому русификация его более трудоемка*.



Рис. 12. Возможности графических шрифтов.

ТЕНЕВОЙ ЭКРАН

*SCREEN, *SWAP

Применение PRO-DOS для создания демонстрационных и учебных программ (например, для видеосопровождения лекций и докладов) предъявляет повышенные требования к скорости вывода картинок и текста на экран монитора. Поскольку система PRO-DOS, как и стандартный Spectrum-Бейсик, является интерпретатором, то она не в состоянии обеспечить высокую скорость выполнения операций обработки экранного изображения. Это приводит к нежелательным задержкам и побочным видеоэффектам, которые могут отвлекать внимание зрителя. Для обхода этих трудностей в PRO-DOS заложена возможность работы с так называемым *теневым экраном.*

Теневой экран — это область ОЗУ, по размерам равная экранной области памяти (6912 байт). Предварительно построенное в ней графическое изображение практически мгновенно можно переместить на экран.

Адрес начала размещения в памяти области теневого экрана задается оператором

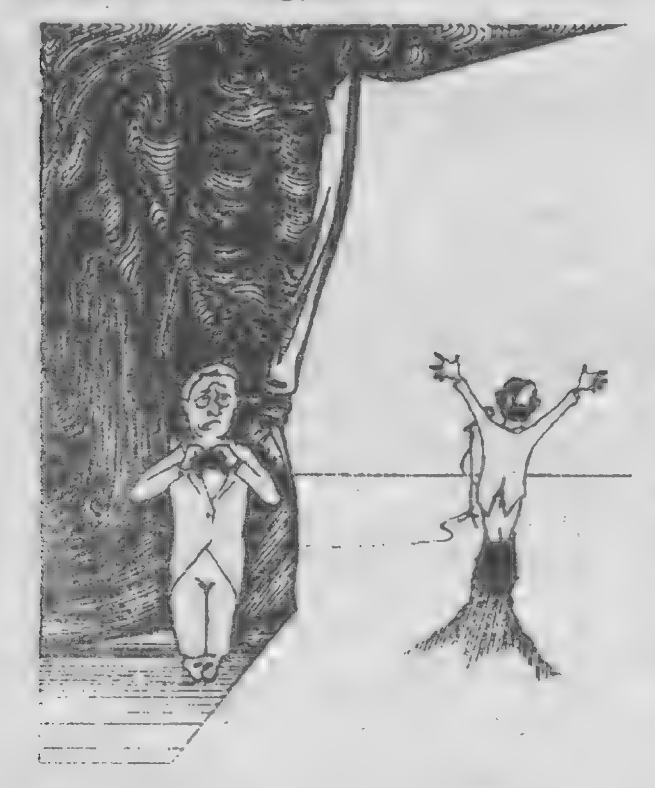
***SCREEN addr**

Параметр **addr** может принимать значения в диапазоне 0...65535, но, как нетрудно догадаться, не все адреса из этого диапазона являются приемлемыми для записи. Хотя ничто не

---

* Автор этих строк ввел в PRO-DOS новый оператор ***CHFNT**, который позволяет переключать все шрифты (стандартный, узкий и графические) с латинского набора на русский и обратно.

помешает посмотреть, как выглядят на экране, скажем, первые 6912 байт ПЗУ.



Теневой экран «признают» все операторы PRO-DOS, а также PRINT и LIST. Инструкции стандартного Бейсика PLOT, DRAW и CIRCLE работают только с «видимым» экраном.

Располагать теневой экран лучше всего вслед за бейсик-программой, защитив ее командой CLEAR addr—1 (то есть установив RAMTOP на единицу меньше адреса начала теневого экрана). Если addr=16384, то области теневого и обычного экранов совпадут. Кстати, при инициализации PRO-DOS устанавливается именно этот режим, и мы не замечаем существования теневого экрана.

Содержимое теневого экрана мгновенно «перебрасывается» в экранную область ОЗУ, то есть становится видимым, с помощью оператора

```
*SWAP
```

При этом содержимое теневой области не изменяется, и там можно продолжать построения. Все неисполняемые операторы не различают теневой и обычный экраны, то есть при переключениях с одного экрана на другой сохраняются описания окон, шрифтов, заливок. Например, определив окно на основном экране

```
*WINDOW5: WSIZE 0,0,10,12
```

– мы «бесплатно» получим точно такое же окно на теневом и наоборот.

Можно пользоваться двумя, тремя и т. д. теневыми экранами (сколько поместится в ОЗУ), переключаясь с одного на другой с помощью оператора *SCREEN. *SWAP будет работать с текущим экраном, то есть с тем, чей адрес был упомянут в последнем выполненном операторе *SCREEN.

Перед использованием теневой экран надо хорошенько «протереть» с помощью *CLEAR или *CLS, поскольку никто не знает, что там находится. Даже если в этой области памяти «сидят» одни нули, то это значит, что фон на всем экране будет «радикального» черного цвета, что не всегда требуется. Приведем пример, демонстрирующий работу с теневым экраном:

```
30 CLEAR 39999: REM Резервируем место под теневой экран
40 *ELLIPSE 80, 80, 70, 30: REM Долго рисуем эллипс на видимом
   экране
50 *SCREEN 40000: REM Переключаемся на теневой экран
60 *CLS: REM Чистим его
70 *ELLIPSE 70, 60, 45, 30: REM Рисуем эллипс на теневом экране
80 *SWAP: REM Мгновенно перебрасываем на видимый экран
```

ПРИЛОЖЕНИЯ

1. Алфавитный перечень операторов PRO-DOS*

Таблица 7.

Оператор	Параметры	Действие	Стр.	Адрес
*BOX	x1, y1, x2, y2	Рисует прямоугольник с координатами вершин (x1, y1) и (x2, y2)	135	62984
*CCHR	—	Переключает шрифт с нормального на узкий	145	62535
*CHR	—	Переключает шрифт с узкого на нормальный	146	62544
*CLEAR	n	Очистка/заливка окна. Параметр устанавливает режим очистки/заливки	144	61559
*CLS	—	Очистка/заливка окна. Эквивалентен *CLEAR 3	145	61643
*CSIZE	n1, n2	Устанавливает размер символов графического шрифта	146	62605
*DIR	n	Устанавливает направление печати для графического шрифта	147	62242
*DRAW	x, y	Рисует отрезок от текущей точки до точки с координатами (x, y)	135	62752
*ELLIPSE	x, y, a, b	Рисует эллипс с координатами центра (x, y) и полуосями a и b	136	63028
*FBOX	x1, y1, x2, y2	Рисует прямоугольник с текстурной заливкой и координатами вершин (x1, y1) и (x2, y2)	138	62841
*FILL	x, y	Осуществляет сплошную заливку замкнутого контура вокруг точки с координатами (x, y)	139	62320
*GPAT	n	Устанавливает шаблон для рисования графических объектов прерывистыми линиями и текстуру для оператора *FBOX	137	62577
*HATCH	x, y, addr	Осуществляет заливку замкнутого контура вокруг точки с координатами (x, y) изображениями символа, закодированными по адресу addr	139	62266
*LARGE	—	Включает режим графического шрифта	147	62559

* В таблице приведен адрес подпрограммы PRO-DOS, реализующей действие оператора.

Оператор	Параметры	Действие	Стр.	Адрес
*LINE	x2, y2	Рисует отрезок прямой линии с координатами концов (x1, y1) и (x2, y2)	135	62896
*MATCH	addr	Заменяет заливку, полученную последним оператором *HATCH, на изображение символа, закодированное по адресу addr	141	62478
*NEW	—	Устанавливает стандартные параметры для всех окон	142	60272
*NOWRAP	—	Разрешает скроллинг при заполнении окна	143	62526
*NORMAL	—	Выключает режим графического шрифта	147	62568
*PAINT	x, y	Производит текстурную заливку замкнутого контура вокруг точки с координатами (x, y)	139	62622
*PLOT	x, y	Рисует точку с координатами (x, y) и устанавливает в ней позицию печати для графических шрифтов	135	61321
*ROLL	m, n	Осуществляет циклический скроллинг окна на одно знакоместо, если m отлично от нуля. Параметр n определяет направление скроллинга	143	62090
*SCREEN	addr	Устанавливает адрес addr начала теневого экрана	148	60255
*SCROLL	m, n	Осуществляет нециклический скроллинг окна на одно знакоместо, если m отлично от нуля. Параметр n определяет направление скроллинга	143	62096
*SWAP	—	Копирует содержимое теневого экрана в экранную область	149	60236
*TPAT	n	Устанавливает шаблон для текстурной заливки окон с помощью операторов *CLEAR и *CLS	144	62591
*TRIANGLE	x1, y1, x2, y2, x3, y3	Рисует треугольник с указанными координатами вершин (x1, y1), (x2, y2) и (x3, y3)	136	62931
*WINDOW	n	Устанавливает текущим окно с номером n	141	60481
*WPOKE	n1, n2	См. Приложение 4	153	60217
*WRAP	—	Блокирует скроллинг при заполнении окна	143	62517
*WSIZE	X1, Y1, X2, Y2	Устанавливает границы текущего окна: (X1, Y1) — координаты левого верхнего угла; (X2, Y2) — правого нижнего (в знакоместах)	141	62693

2. Распределение памяти при работе с PRO-DOS

P_RAMT (23732)*		
	Символы, определяемые пользователем	
UDG (23675)		
	Свободная область	
		63788
	Интерпретатор PRO-DOS	
		60000
	Свободная область	
RAMTOP (23730)		
	...	
E_LINE (23641)		
	Переменные Бейсика	
VARS (23627)		
	Бейсик-программа	
PROG (23635)		

3. Вывод на принтер

При работе в среде PRO-DOS возникают определенные сложности с выводом на принтер. Выполняемая при инициализации PRO-DOS модификация канала "P" (см. стр. 133) дает о себе знать при попытке обращения к принтеру: вместо бумаги вывод идет на экран.

PRO-DOS обычно используют для написания демонстрационных программ, как правило, не требующих работы с принтером. Но в некоторых случаях, например, при отладке программы бывает полезно распечатать ее листинг или значения переменных. Листинг проще всего получить, воспользовавшись командой LLIST в момент, когда PRO-DOS еще не инициализирована оператором RANDOMIZE USR 60000. Например, для распечатки текста фирменной демонстрационной программы PRODOSDEMO можно загрузить ее в память с помощью оператора MERGE (во избежание автоматического запуска программы) и выполнить LLIST.

Однако описанный метод не позволяет выводить информацию на принтер непосредственно в процессе работы программы. Эту проблему можно решить, если перед операцией вывода на принтер модифицировать две ячейки памяти, отвечающие за работу канала "P", так, чтобы перенаправить действие операторов печати LPRINT и LLIST с экрана на принтер. А затем, сразу после печати, вернуть содержимое этих ячеек в исходное (характерное для PRO-DOS) состояние. Указанные ячейки располагаются в области информации о каналах** по адресам 23749 и 23750

* В скобках приведены адреса системных переменных, в которых записаны значения адресов соответствующих границ областей памяти.

** Подробнее см. в [1].

(если подключен и инициализирован Beta-Disk Interface, то по адресам 23861 и 23862). В этих ячейках должен содержаться адрес процедуры (драйвера), обслуживающей канал "P". При инициализации PRO-DOS в них записывается значение 60514. Перед работой с принтером это значение надо заменить на адрес применяемого драйвера (первая ячейка должна содержать младший байт адреса, а вторая — старший).

Если Вы владелец фирменного ZX Spectrum с «родным» для него принтером ZX-Printer, то нужный драйвер «зашит» в ПЗУ по адресу 2548 (#09F4). В этом случае программа, использующая вывод на принтер, может выглядеть так (предположим, что дисковый интерфейс отсутствует):

```
 10 RANDOMIZE USR 60000
 20 OPEN #2, "P"
 ...
100 LET a=100: LET b=200
 ...
200 POKE 23749,244: POKE 23750,9: REM Размещаем младший и
    старший байты числа 2548
210 LPRINT a: LPRINT b: REM Любые операции вывода на принтер
220 POKE 23749,98: POKE 23750,236: REM Размещаем младший и
    старший байты числа 60514
 ...
```

Программа выведет значения переменных a и b на принтер.

Если у Вас самодельный Spectrum-совместимый компьютер с нестандартным принтером, то нужно вместо значения 2548 подставить адрес размещения этого драйвера. Чаще всего драйверы располагают в ОЗУ в области буфера принтера по адресу 23296. Если это так, то достаточно заменить строку 200 в приведенном примере на следующую (для разнообразия примем, что Вы еще и обладатель интерфейса Beta-Disk, тогда изменится и 220-я строка):

```
200 POKE 23861,0: POKE 23862,91: REM Размещаем младший и
    старший байты числа 23296
220 POKE 23861,98: POKE 23862,236
```

Разумеется, строки с номерами 200 и 220 можно оформить в виде двух подпрограмм и вызывать их каждый раз при необходимости вывода на принтер.

4. Дополнительные сведения об окнах

PRO-DOS располагает еще одним оператором — *WPOKE. Он меняет содержимое внутренних системных переменных PRO-DOS, описывающих текущие параметры окон. Выполнение

```
*WPOKE n, m
```

эквивалентно действию оператора

```
POKE 60321+ABS(n), ABS(m)
```

– где n и m — целые числа в диапазоне –255...255.

Внутренние системные переменные расположены в области ОЗУ, начиная с адреса 60321. Для каждого окна, включая нулевое, в этой области отведено по 20 байтов. Объем всей области переменных равен 8×20=160 байт.

Оператор

*WPOKE 20*w+(v−1), k

запишет число k в v-ю ячейку области системных переменных w-го окна.

Назначение ячеек:

1...4 — размеры окна. Устанавливаются оператором *WSIZE. Начальные значения — 0, 0, 31 и 23;

5 и **6** — координаты текущей позиции печати (номер столбца, номер строки). Начальные значения — 0 и 0;

7 и **8** — координаты последней выставленной точки. Изменяется только при выполнении графических операторов PRO-DOS. Начальные значения — 0 и 191;

9 — тип заливки окна. Устанавливается оператором *TPAT. Начальное значение — 0;

10 — тип линии. Устанавливается оператором *GPAT. Начальное значение — 255;

11 и **12** — размер символов графического шрифта. Изменяются оператором *CSIZE. Начальные значения — 1 и 1;

13 — тип шрифта и разрешение/запрет скроллинга: если установлен режим *NOWRAP, то значение байта равно 128 для стандартного шрифта, 160 — для графического и 192 — для узкого. Если установлен режим *WRAP, приведенные значения увеличиваются на 4. Начальное значение — 128. (Не следует помещать в эту ячейку неспецифицированные значения — это приводит к непредсказуемым результатам);

14 — направление печати. Устанавливается оператором *DIR. Содержимое ячейки не совпадает со значениями параметра *DIR: 1 — вправо, 2 — вниз, 4 — вверх и 8 — влево, диагональные направления указываются суммой соответствующих значений горизонтального и вертикального. Начальное значение — 1;

15 — постоянные цветовые атрибуты окна. Значение определяется по формуле, приведенной на стр. 55. Начальное значение — 56 (INK 0; PAPER 7; BRIGHT 0; FLASH 0);

16 — не используется;

17 — атрибуты OVER и INVERSE. Значение определяется по формуле

k = <OVER> + 4 × <INVERSE> + 32 × G

- где G = 1, если атрибуты после инициализации окна хотя бы раз изменялись, иначе G = 0. Начальное значение — 0;

18 и **19** — адрес размещения теневого экрана; устанавливается оператором *SCREEN; первоначальное значение — 16384 (стандартная экранная область);

20 — не используется.

Недостатком оператора *WPOKE является отсутствие «защиты от дурака»: если первый аргумент превышает 159, то *WRAP будет изменять содержимое ячеек памяти за пределами таблицы «оконных» переменных. Это может привести к чему угодно, вплоть до сброса компьютера.

LASER BASIC

Laser Basic* — это целый пакет программ, предназначенный для манипулирования графическим изображением. С его помощью можно выводить на экран монитора картинки (спрайты), инвертировать их, поворачивать, зеркально отображать... и, что самое ценное, самое нужное, самое приятное — перемещать по экрану. Довольно быстро и без особых ухищрений можно создавать мультфильмы и даже динамические игры.

Картинки подготавливаются в специальной программе — **генераторе спрайтов**, входящей в пакет Laser Basic. Можно воспользоваться и любым графическим редактором для ZX Spectrum (Art Studio, Artist II) или даже позаимствовать готовые картинки из фирменных программ.

Программы изначально пишутся в **интерпретаторе** Laser Basic, который распознает и выполняет более ста новых операторов и функций. Отладив программу в интерпретаторе, можно затем обработать ее с помощью **компилятора** Laser Basic. Это значительно (примерно в 2 раза) увеличивает скорость работы программы, уменьшает занимаемую ею память и защищает ее текст от просмотра.

В пакет входят также две демонстрационные программы: **Demo** и **Game**, задача которых показать все достоинства Laser Basic.

ТЕРМИНОЛОГИЯ

Основные объекты, которыми оперирует Laser Basic, — это окна и спрайты.

Окно экрана выделяет прямоугольную область, на которую распространяется действие операторов, выполняющих графические

---

* Автор Kevin Hambleton, фирма Oasis Software, версия 1.3, спецификация файлов приведена в Приложении 5.

преобразования. Окно задается с точностью до знакоместа координатами левого верхнего угла, шириной и высотой.

Другой объект, с которым имеет дело Laser Basic, — это спрайт. Если окно привязывается к некой области экрана (причем безразлично, что в этой области изображено), то спрайт, наоборот, имеет собственный внешний вид и может «гулять» по экрану. (Все прыгающие, летающие, катящиеся и прочие двигающиеся изображения в компьютерных играх и есть те самые спрайты, о которых идет речь.) *Спрайт* (от англ. *sprite* — эльф) можно определить как перемещаемый графический объект с неизменными рисунком и размером. Спрайты создаются обычно до написания программы и хранятся в памяти компьютера в виде так называемого *спрайт-файла*. Спрайт-файл может объединять до 255 спрайтов разного размера (от знакоместа и более). По желанию программиста спрайты могут быть вызваны из спрайт-файла и помещены на экран, где с ними можно делать все, что позволит Laser Basic, и, конечно, передвигать. Для чего спрайты и предназначены.



Кроме термина «окно экрана» (или просто — окно) в лексиконе Laser Basic есть еще и такое понятие, как *окно спрайта*, обозначающее прямоугольную область спрайта. Выделение в спрайте окна позволяет трансформировать, а также выводить на экран не только целиком весь спрайт, но и часть его, что бывает очень полезно.

ИНТЕРПРЕТАТОР

Интерпретатор Laser Basic не накладывает никаких ограничений на использование операторов стандартного Spectrum-Бейсика. В этом плане единственным его недостатком можно считать то, что он «затирает» область символов, определяемых пользователем*.

Операторы Laser Basic задаются ключевыми словами, состоящими из пяти символов: первый — точка (.), четыре последующих — прописные (и только прописные) буквы латинского алфавита. Например: .PTBL, .MOVE. Набираются ключевые слова посимвольно.

При вводе строки программы интерпретатор Laser Basic проверяет ее на правильность синтаксиса. Строка с ошибкой «не вводится», а в сомнительном месте обычно появляется мерцающий знак вопроса (?).

---

* Впрочем, этот недостаток легко устраняется изменением системной переменной UDG (23675/76), см. стр. 64.

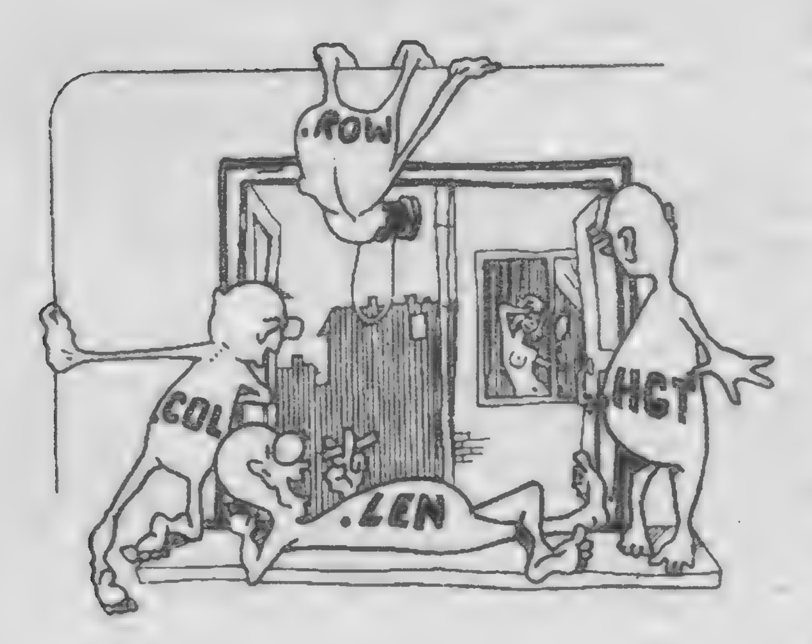
Параметры операторов Laser Basic передаются интерпретатору иначе, чем в стандартном Бейсике и других его диалектах, — не указанием значений вслед за ключевым словом, а через специальные *графические переменные*. Перед выполнением оператора необходимо присваивать требуемые значения определенным графическим переменным*.

Имена графических переменных состоят из четырех символов: первый — точка, остальные — строчные буквы латинского алфавита.

Всего графических переменных десять:

.ROW, .COL – вертикальная и горизонтальная координаты в знакоместах (пределы, соответственно, 0...23 и 0...31);

.HGT, .LEN – высота и ширина графического объекта (окна экрана, спрайта) в знакоместах (0...24 и 0...32);



Строго говоря, переменные .ROW, .COL, .HGT и .LEN могут принимать любые значения от 0 до 255, в том числе и отрицательные. Однако в этом случае может возникнуть ситуация, когда спрайт или окно частично или полностью выйдет за пределы экрана. Кроме того, надо помнить, что положительные значения, превышающие 127, будут интерпретироваться как отрицательные и, наоборот, отрицательные в пределах −128...−255 — как положительные. Например, 255 приравнивается к −1, а −251 приравнивается к 5.

.SRW, .SCL – вертикальная и горизонтальная координаты внутри спрайта в знакоместах;

.NPX – величина и направление вертикального скроллинга в пикселях (−128...127);

.SPN – номер спрайта в спрайт-файле (1...255);

.SP1, .SP2 – номера первого и второго спрайтов (1...255) для операторов, работающих с двумя спрайтами.

В отличие от переменных стандартного Бейсика графические переменные всегда определены: после загрузки Laser Basic их значения равны нулю.

Присвоение значений графическим переменным происходит без использования оператора LET. Например:

.ROW=1 или .COL=c или .LEN=s+d*3

---

* Из этого правила выпадают только несколько операторов.

Графические переменные не могут быть параметрами операторов, аргументами функций Spectrum-Бейсика или частью выражений. Нельзя, например, писать:

```
LET a=.ROW или PRINT .COL или .SPN=.COL
```

Для присвоения значений графических переменных обычным числовым переменным используются специальные *графические функции*. Имена их идентичны именам соответствующих графических переменных, только вместо точки в них ставится знак вопроса (например, ?ROW, ?COL).

Графические функции могут использоваться только с оператором присвоения в строках типа:

```
LET r=?ROW
```

– где r — обычная числовая переменная, ?ROW — функция, возвращающая значение графической переменной .ROW.

Аналогично, с помощью соответствующих графических функций (?COL, ?LEN, ?HGT, ?SP1, ?SP2, ?SPN, ?NPX, ?SCL, ?SRW) числовым переменным могут быть присвоены значения остальных девяти графических переменных.

Нужно учитывать, что если какая-либо графическая переменная принимает отрицательное значение, то результат выполнения соответствующей функции окажется положительным числом больше 127. Например:

```
10 .COL=–5: LET Z=?COL
20 PRINT Z: REM Z равна не –5, а 251 (256–5)
```

Загрузка и запуск интерпретатора

Фирменный вариант пакета Laser Basic предусматривает возможность работы всех входящих в него программ с лентой и микродрайвом*. По ряду причин микродрайв в нашей стране не получил распространения. Поэтому довольно быстро появилась версия, поддерживающая работу и с магнитофоном, и с диском в системе TR-DOS. Именно о ней и пойдет речь в предлагаемом описании.

Все примеры программ, приведенные в описании, кроме специально оговоренных, будут работать и с лентой, и с диском. Если Вы пользуетесь дисководом, то перед операторами LOAD, SAVE, MERGE и VERIFY следует вставлять RANDOMIZE USR 15619: REM:

В пакет Laser Basic включена программа-диспетчер **Laser**. Загрузим ее, и через некоторое время на экране появится меню (рис. 13) (для ленточной версии пятая и шестая позиции будут

---

* Микродрайв – накопитель на кольцевой магнитной ленте.

несколько отличаться от изображенных на рисунке). Рассмотрим позиции меню (опции):

1 EXECUTE LASER BASIC

Запуск интерпретатора Laser Basic. При выборе этой опции бейсик-программа «сбросится» и в ОЗУ останутся только коды интерпретатора. После этого можно выполнять операторы Laser Basic, загружать и сохранять файлы и т. д.;

2 LOAD YOUR OWN SPRITES

Загрузка спрайт-файла, созданного пользователем. На запрос INPUT SPRITE START ADDRESS нужно ввести адрес загрузки (нижнюю границу) спрайт-файла*;

3 LOAD "SPRITE2A" SPRITES

Загрузка спрайт-файла SPRITE2A;

4 LOAD "SPRITE2B" SPRITES

Загрузка спрайт-файла SPRITE2B (этот файл используется в демонстрационной программе Demo);

```
LASER BASIC © OASIS SOFTWARE

BETA-DISK VERSION
1....EXECUTE LASER BASIC
2....LOAD YOUR OWN SPRITES
3....LOAD "SPRITE2A" SPRITES
4....LOAD "SPRITE2B" SPRITES
5... PRINT CATALOGUE
6....RETURN TO TAPE MENU

PLEASE SELECT OPTION.
```

Рис. 13. Главное меню программы LASER.

5 SAVE LASER BASIC TO 5'25 FLOPPY DISK

либо

5 PRINT CATALOGUE

Копирование Laser Basic с ленты на диск в системе TR-DOS (для «ленточного» меню) или **выдача каталога диска** (для «дискового» меню);

6 RETURN TO TAPE MENU либо **RETURN TO DISK MENU**

Переход из ленточного меню в дисковое и обратно.

Вывод спрайтов на экран

Загрузим один из спрайт-файлов, например, фирменный файл SPRITE2B** (опция 4 меню), запустим интерпретатор Laser Basic (опция 1) и выведем на экран один из спрайтов, например, первый. Для задания номера спрайта и его местоположения на экране

---

* Подробнее см. в подразделе «Изменение размеров области спрайт-файла» и разделе «Создание спрайтов».

** Во всех примерах данного описания предполагается, что загружен именно этот спрайт-файл.

(координат верхнего левого угла) воспользуемся соответствующими графическими переменными:

```
10 .SPN=1: REM Зададим номер спрайта
20 .ROW=10:.COL=10: REM Определим место в 10-й строке
   и 10-м столбце экрана
```

Теперь можно выполнить самый простой и самый популярный оператор Laser Basic **.PTBL**:

```
30 .PTBL REM Поместим спрайт на экран
```

Вывод на экран части (окна) спрайта

Если нет необходимости переносить на экран весь спрайт, можно вывести лишь его часть — окно спрайта. Делает это оператор **.PWBL**. Кроме номера спрайта и координат размещения изображения на экране, для работы этого оператора следует задать размеры и координаты окна спрайта, то есть присвоить требуемые значения переменным: .HGT, .LEN — высота и ширина окна, .SCL — левая колонка, .SRW — верхняя строка окна спрайта относительно границ спрайта:

```
10 .SPN=1:.ROW=15:.COL=15
20 .HGT=2:.LEN=3: REM Размеры окна спрайта
30 .SCL=1:.SRW=0: REM Координаты окна спрайта
40 .PWBL
```

Перенос атрибутов

Описанные выше операторы вывода спрайтов по умолчанию переносят на экран и их атрибуты. Но можно и запретить перенос атрибутов, выполнив оператор **.ATOF**. Вновь разрешает перенос оператор **.ATON**. Проиллюстрируем на примере:

```
10 .ROW=0:.COL=0:.SPN=4:.PTBL: REM Поместим спрайт на экран
20 .ATOF: REM Запретим перенос атрибутов
30 .COL=8:.PTBL: REM Выведем «черно-белое» изображение
40 .ATON: REM Вновь разрешим перенос атрибутов
50 .COL=16:.PTBL: REM Выведем как прежде — в цвете
```

Следует учитывать, что операторы .ATON и .ATOF воздействуют не только на вывод спрайтов, но и на все операции типа «спрайт—экран» и «экран—спрайт», которые будут описаны ниже.

Преобразование окна экрана

Перед тем, как начать гонять спрайт, помещенный на экран, из угла в угол, поэкспериментируем с ним, не сдвигая с места. Laser Basic обладает широкими возможностями трансформации

картинок*. И, что ценно, можно преобразовывать не только все экранное изображение, но и его часть, в пределах заданного окна экрана.

Расположение и размеры окон определяются четырьмя графическими переменными: .ROW, .COL, .HGT и .LEN. Если окно выходит за пределы экрана, оно автоматически обрезается по его размеру.

Операторы, выполняющие графические преобразования, легко узнать по букве V (Video), завершающей ключевое слово.

Простейший из операторов, преобразующих картинку,— **.INVV**. Он инвертирует изображение в пределах установленного окна экрана, то есть меняет цвет фона на цвет тона и наоборот.

Оператор **.MIRV** выполняет зеркальное отображение картинки в окне относительно его вертикальной оси симметрии. Действие этого оператора распространяется только на изображение и не затрагивает раскраску (атрибуты) окна. Атрибуты отображаются специальным оператором **.MARV**. Приведем пример:

```
10 .ROW=1:.COL=1:.LEN=14:.HGT=1: REM Зададим окно
20 PRINT AT 1,1; INK 1;"LEFT": REM Поместим в окно текст
30 PRINT AT 1,10; INK 2;"RIGHT"
40 PAUSE 100:.MIRV: REM Отобразим текст без атрибутов
50 PAUSE 100:.MARV: REM Теперь отобразим атрибуты
```

Laser Basic позволяет уже после создания экранного изображения перекрашивать его части. Оператор **.SETV** присваивает всем знакоместам окна текущие (постоянные) атрибуты.

Ну и, конечно же, можно очистить окно экрана: выполняет это действие оператор **.CLSV**. Он похож на стандартный CLS, только действует не на весь экран, а на его часть — окно, и, кроме того, не изменяет атрибутов знакомест:

```
10 .HGT=3:.LEN=6:.ROW=4: REM Задание окна
20 PRINT AT 5,1;"RED";AT 5,6;"YELLOW": REM Вывод в окно
30 .COL=6: PAPER 6:.SETV: REM Изменение цвета фона в окне
40 .COL=0: PAPER 2:.SETV: REM Изменение цвета фона в окне
50 .LEN=12: PAUSE 100:.INVV: REM Инверсия окна
60 PAUSE 100:.CLSV: REM Очистка окна
```

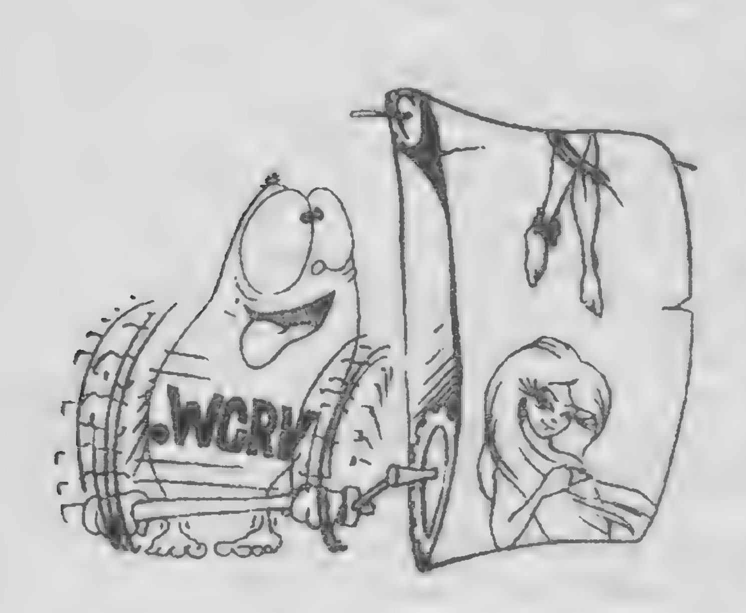
Следующий блок операторов, занятых преобразованием изображения, — это операторы, смещающие картинки в окнах экрана на заданное число пикселей, то есть производящие скроллинг. Laser Basic обеспечивает как простой скроллинг, при котором изображение, вытесненное за пределы окна, бесследно пропадает, так и циклический, выводящий смещенное за границу окна изображение с противоположной стороны.

Горизонтальный скроллинг изображения (без атрибутов) может быть выполнен вправо или влево на фиксированный шаг: 1, 4 и 8 пикселей. Операторы горизонтального скроллинга **.SL1V**, **.WL1V**,

---

* Здесь картинка — не только помещенный на экран спрайт, но и построения, выполненные с помощью графических операторов Spectrum-Бейсика, тексты, выведенные оператором PRINT и пр.

.SL4V , **.WL4V**, **.SL8V** , **.WL8V**, **.SR1V**, **.WR1V**, **.SR4V**, **.WR4V**, **.SR8V** и **.WR8V** по отдельности описаны в Приложении 1. Но и без подсказки можно легко разобраться, «кто есть кто»: первая буква ключевого слова сообщает о виде скроллинга (S — нормальный, W — циклический), вторая — о направлении (L — левый, R — правый), а цифра — о шаге (1, 4 и 8 пикселей).



Операторов, отвечающих за вертикальный скроллинг изображения, в Laser Basic лишь два: **.SCRV** выполняет простой скроллинг, **.WCRV** — циклический. Мало? Вполне достаточно, поскольку величина и направление смещения при вертикальном скроллинге задаются специальной графической переменной .NPX. Она может принимать значения от −128 до 127. Модуль значения задает шаг скроллинга в пикселях. Направление определяется знаком: положительное значение соответствует скроллингу вверх, отрицательное — вниз.

Горизонтальный и вертикальный скроллинг атрибутов (на целое знакоместо) выполняется операторами **.ATLV** — влево и **.ATRV** — вправо, **.ATUV** — вверх, **.ATDV** — вниз. Правда, обеспечивают они лишь циклический скроллинг.

В операциях вертикального скроллинга задействована область буфера принтера (см. [1]). И так как ее объем ограничен 256 байтами, то вертикальный скроллинг возможен лишь, если результат перемножения значений графических переменных .NPX и .LEN не превышает 256 (в противном случае может произойти сбой программы). Непосредственно в программе проверить это условие можно следующей строкой:

```
LET X=?NPX: LET Y=?LEN: IF ABS (X*Y) <= 256 THEN .WCRV
```

Наборы переменных

Специальная переменная .SET позволяет, единожды задав *набор переменных*, в последующем присваивать целой группе графических переменных нужные значения указанием только номера набора. Задаются наборы переменных строками типа:

```
10 .SET=1:.HGT=5:.LEN=5:.ROW=0:.COL=0: REM Окно 1
20 .SET=2:.HGT=4:.LEN=5:.ROW=5:.COL=7:.NPX=1: REM Окно 2 и
   параметр вертикального скроллинга .NPX
```

В данном примере наборам графических переменных присвоены номера 1 и 2. Всего в программе может быть определено 16 наборов (от 0 до 15).

Теперь, чтобы выполнить, например, горизонтальный скроллинг 1-го окна и вертикальный 2-го окна, достаточно задать:

```
50 .SET=1:.WR4V:.SET=2:.WCRV
```

После выполнения этой строки графические переменные будут иметь значения, соответствующие последнему вызванному набору номер 2.

Последний указанный в программе набор переменных становится *текущим*. При изменении графических переменных в процессе работы программы модифицируется текущий набор (остальные наборы не меняются).

Перемещение спрайтов (1-й способ)

От преобразования картинки в статичном окне до движения спрайта — один шаг. Поместим на экран спрайт. Зададим окно, по размерам покрывающее спрайт, и произведем скроллинг. Спрайт переместится. Как уже упоминалось, при движении по горизонтали шаг может быть 1, 4 и 8 пикселей, по вертикали — произвольный. Например, это может выглядеть так (рис. 14):

```
10 .SPN=4:.ROW=0:.COL=0
20 .PTBL: REM Вывод спрайта на экран
30 .LEN=30:.HGT=4: REM Окно
40 FOR N=1 TO 240
50 .WR1V: REM Скроллинг вправо на один пиксель
60 NEXT N
```

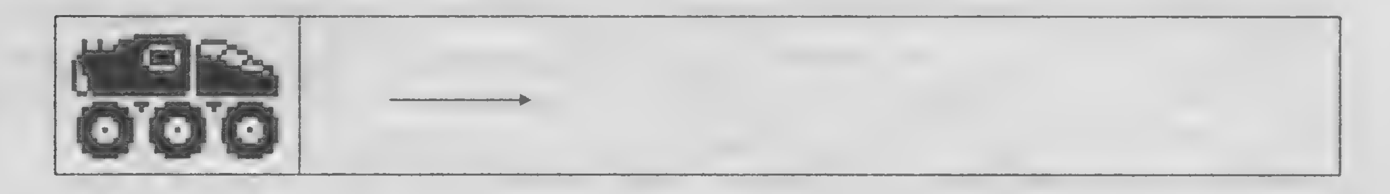


Рис. 14. Перемещение спрайта путем скроллинга окна.

Существенным недостатком такого способа «оживления» спрайтов является то, что перемещение происходит вместе с фоном*. Ведь скроллинг захватывает все изображение в окне, не разбирая, спрайт ли это или что-то иное.

Перемещение спрайтов (2-й способ)

Этот способ лишен недостатков предыдущего. Он основан на последовательном вызове спрайта на экран оператором .PTBL с изменением на единицу значений .ROW и .COL. Меняя одну из переменных, заставим спрайт двигаться параллельно границам

* В данном случае словом «фон» мы называем любое изображение, находящееся на экране.

экрана, меняя обе — по диагонали. Спрайт, перемещаемый таким способом, должен иметь по краям пустое пространство шириной в одно знакоместо. Иначе спрайт, помещенный на экран в предыдущем шаге, не будет затираться следующим выводимым спрайтом, и по экрану протянется след.

Спрайт, изображенный на рис. 15а, можно двигать во всех направлениях. Если же заранее известно, что спрайт будет перемещаться, например, только налево, то для скроллинга достаточно оставить справа пустую колонку, шириной в одно знакоместо (рис. 15б).



Рис. 15. Подготовка спрайта к перемещению.

Избежав одной неприятности, возникающей при перемещении спрайта скроллингом (смещение фона), мы получили другую — спрайт, перемещаемый с помощью оператора .PTBL, затирает фон.

Справиться с проблемой сохранения фона при перемещении спрайтов можно, только умело пользуясь операторами наложения спрайта на экран по различным принципам.

Наложение спрайтов

Laser Basic позволяет помещать спрайт на экран не только примитивным способом, затирая фон (оператор .PTBL), но и по-разному совмещая картинку спрайта и экранное изображение. Принципов совмещения (наложения) три, и обозначаются они по названиям логических операций: OR, AND и XOR. Реализуют эти принципы соответственно три оператора:

.PTOR – объединяет картинки (остаются включенными пиксели, которые были включены либо на экране, либо в спрайте);

.PTND – оставляет только совпавшее изображение: пиксели, включенные одновременно и на экране, и в спрайте;

.PTXR – стирает совпавшее изображение, оставляя все остальное: пиксели, включенные либо только на экране, либо только в спрайте.

Проиллюстрируем действие этих операторов программой, выполняющей наложение спрайта номер 5 на спрайт номер 1:

```
10 .ROW=1:.SPN=1
20 FOR n=0 TO 32 STEP 8:.COL=n:.PTBL: NEXT n
30 PAUSE 50:.SPN=5
40 .COL=0:.PTBL
50 .COL=8:.PTOR
60 .COL=16:.PTXR
70 .COL=24:.PTND
```

Приведенный рисунок (рис. 16) поясняет принципы наложения спрайтов.

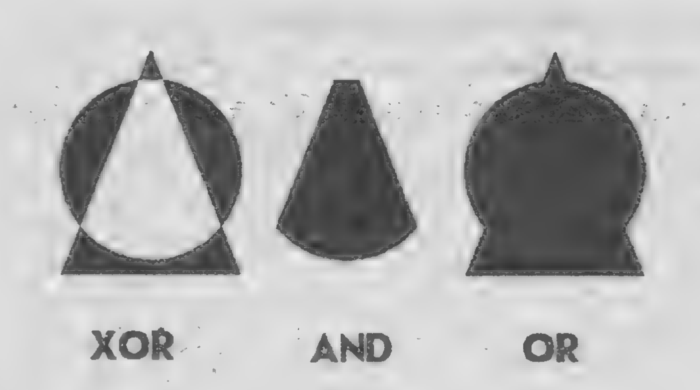


Рис. 16. Наложение спрайтов по различным принципам.

Кроме операторов наложения целых спрайтов в Laser Basic имеются инструкции, осуществляющие те же действия с окнами спрайтов: **.PWOR**, **.PWND** и **.PWXR**. Действуют они аналогичным образом, но предварительно должны быть определены переменные .SRW, .SCL, .HGT и .LEN, задающие положение и размеры окна в спрайте. Присваивая значения этим переменным, нужно следить, чтобы окно не выходило за пределы спрайта, иначе команда не пройдет, хотя сообщения об ошибке и не появится. Если добавить в предыдущий пример строку

```
35 .SRW=0:.SCL=0:.HGT=2:.LEN=2
```

и в строках с 40-й по 70-ю заменить буквы PT на PW, то получившаяся программа проиллюстрирует действие операторов наложения окна спрайта на экран.

Перемещение спрайтов (3-й способ)

После ознакомления с принципами наложения изображения можно рассмотреть и способы перемещения спрайтов с сохранением фона.

Например, можно предложить следующую последовательность действий:

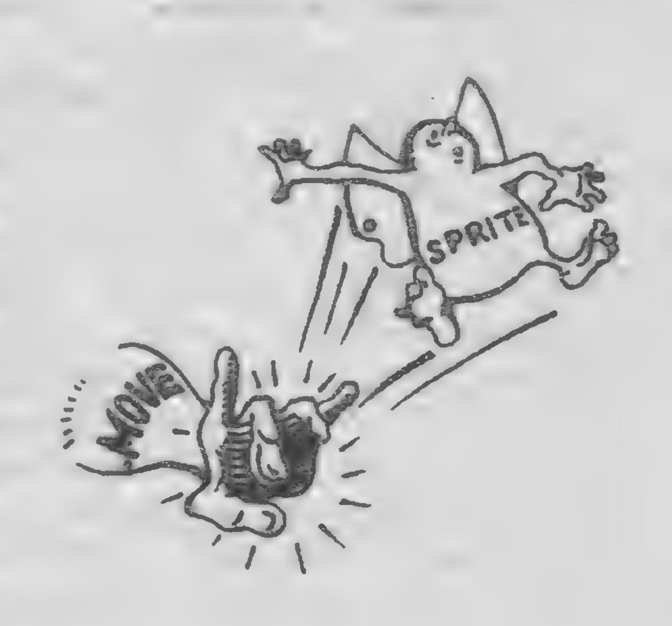
1. поместить спрайт на экран оператором .PTXR. Фон при этом сохранится, но сам спрайт немного пострадает: в точках пересечения фона и спрайта образуются «прорехи» (во многих случаях это допустимо);
2. через требуемое время стереть спрайт, вторично выполнив оператор .PTXR с теми же значениями графических переменных. Фон полностью восстановится;
3. изменить переменные .ROW и .COL и поместить спрайт в новое место опять же оператором .PTXR. И так далее.

Можно, конечно, проверить работу приведенного алгоритма, но, вообще-то, в этом нет необходимости, так как он реализован в специально предназначенном для перемещения спрайтов операторе **.MOVE**.

.MOVE работает следующим образом: накладывает по принципу XOR спрайт с номером, заданным графической переменной .SP2, на экран, соответственно, переменным .ROW и .COL. Потом, изменив значения переменных .ROW и .COL на величины смещения, заданные соответственно переменными .HGT и .LEN, накладывает по тому же принципу XOR на новое место спрайт с номе-

ром .SP1. В завершение переменные .SP1 и .SP2 обмениваются значениями.

Для перемещения спрайта с помощью оператора .MOVE переменным .SP1 и .SP2 присваивают одинаковые значения, то есть задают один спрайт*. Затем помещают оператором .PTXR спрайт с этим номером в исходную точку, заданную .ROW и .COL. На первом шаге выполнения .MOVE спрайт будет стерт, на втором — помещен на другое место. Далее можно еще и еще раз выполнять .MOVE без каких-либо дополнительных действий, если, конечно, не потребуется изменить скорость и направление движения спрайта. Делается это присвоением переменным .HGT и .LEN других значений. Приведем пример:




```
10 .SP1=5:.SP2=5: REM Перемещать будем спрайт с номером 5
20 .ROW=0:.COL=0:.SPN=5:.PTXR: REM Поместим его на экран
30 .HGT=1:.LEN=1: REM Зададим смещение
40 FOR n=1 TO 15:.MOVE: PAUSE 4: NEXT n: REM Сдвинем на 15
   знакомест по диагонали
```

Копирование изображения с экрана в спрайт

В Laser Basic имеется группа операторов, которые копируют изображение с экрана в спрайт (или окно спрайта), то есть помещают изображение в спрайт-файл, находящийся в памяти. Самый употребительный из них — **.GTBL**, он переносит изображение из окна экрана в спрайт с номером, заданным переменной .SPN. Остальные операторы этой группы описаны в Приложении 1.

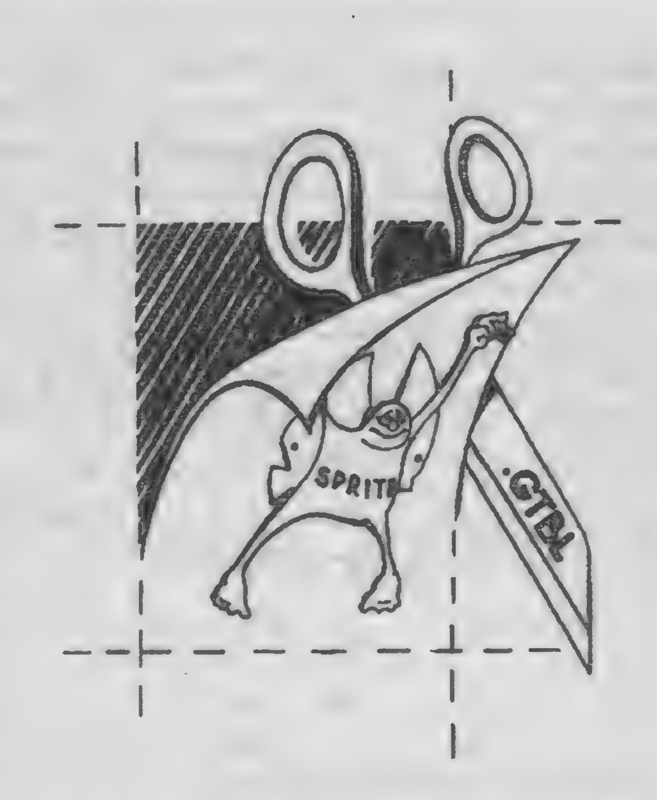
Перемещение спрайтов (4-й способ)

Используя оператор копирования изображения из окна экрана в память .GTBL, можно реализовать еще один способ перемещения спрайтов с сохранением фона. Суть его проста: окно экрана, в которое будет помещен спрайт, сохраняется оператором .GTBL в буфере (пустом спрайте), после этого предназначенный для перемещения спрайт выводится на экран оператором .PTOR, а через некоторое время (в зависимости от требуемой скорости перемещения) изображение из буфера копируется обратно на экран, восстанавливая фон. Далее в буфер помещается изображение из

---

* Можно использовать и разные спрайты, получая, например, «прыгающий» спрайт, но это уже высший пилотаж.

окна, соответствующего новому положению спрайта, спрайт выводится в это окно и т. д. Пустой спрайт-буфер, по размерам равный перемещаемому спрайту, создается при формировании спрайт-файла.



Поскольку для примеров мы пользуемся готовым спрайт-файлом SPRITE2B, то не будем формировать пустой спрайт, а «пожертвуем» спрайтом .SPN=2 — используем его в качестве буфера. Размеры его равны размерам спрайта .SPN=1. Следующая программа перемещает спрайт .SPN=1 слева направо на 10 знакомест с сохранением фона:

```
10 .ROW=0:.LEN=4:.HGT=2: REM Окно в размер спрайта номер 1
20 .COL=5:.SPN=3:.PTBL: REM Фоном служит спрайт номер 3
30 FOR N=0 TO 10:.COL=N
40 .SPN=2:.GTBL: REM Копирование окна с фоном в буфер
50 .SPN=1:.PTOR: REM Вывод спрайта на экран
60 PAUSE 10
70 .SPN=2:.PTBL: REM Восстановление окна
80 NEXT N: REM Следующий шаг
```

Преобразование спрайтов

Операторы преобразования спрайтов по своему действию аналогичны инструкциям преобразования окна экрана (см. стр. 160), но производят изменения не на экране, а в памяти. Для того чтобы увидеть действие этих операторов, нужно после их выполнения поместить измененный спрайт на экран, например, оператором .PTBL.

Операторы этой группы можно распознать по букве M на конце ключевого слова (от англ. *Memory* — память).

Не извлекая спрайт из памяти, его можно инвертировать (**.INVM**), зеркально отображать относительно вертикальной оси симметрии (**.MIRM**), а также зеркально отображать атрибуты спрайта (**.MARM**). Предварительно необходимо задать номер преобразуемого спрайта — присвоить соответствующее значение переменной .SPN

Во всех знакоместах спрайта можно установить текущие атрибуты (**.SETM**). А можно и полностью очистить спрайт оператором **.CLSM**.

Операторы, производящие скроллинг спрайтов в памяти, аналогичны операторам скроллинга окон экрана. Например, **.WR1M** производит циклический скроллинг спрайта в памяти вправо на 1 пиксель. Нужно только задать номер спрайта в переменной .SPN. Операторы скроллинга спрайтов можно найти в Приложении 1.

Для работы со спрайтами в памяти существуют и операторы, которым нет «экранных» аналогов. Например, можно повернуть спрайт на 90°. Оператор **.SPNM** копирует спрайт с номером .SP1 в спрайт .SP2* с поворотом изображения на 90° по часовой стрелке. Предварительно второй спрайт (.SP2) должен быть очищен. Особо следует обратить внимание на то, чтобы высота второго спрайта равнялась ширине первого, а ширина, соответственно, — высоте.

Оператор **.DSPM** позволяет прямо в процессе работы программы получить в спрайте с номером .SP1 увеличенное в четыре раза изображение спрайта .SP2. Естественно, что для успешного выполнения этой операции высота и ширина спрайта .SP1 должны быть в два раза больше, чем у исходного .SP2. Спрайт .SP1 предварительно должен быть очищен.

Наложения «спрайт—окно спрайта»

Операторы взаимодействия спрайта с окном другого спрайта (напомним, что эти преобразования происходят в памяти) во многом похожи на соответствующие инструкции для окон экрана. Ключевые слова операторов этой группы начинаются с букв GM и PM (от англ. *Get Memory* и *Put Memory*). Например, операторы **.GMBL**, **.GMOR**, **.GMXR** и **.GMND** переносят или накладывают по различным принципам спрайты на окна в других спрайтах, а **.PMBL** **.PMOR**, **.PMXR**** и **.PMND**, наоборот, окно спрайта переносят в «целый» спрайт. Номер «целого» спрайта заносится в переменную .SP1, номер спрайта с окном — в .SP2. Размеры окна в спрайте .SP2 не задаются: подразумевается, что они равны размерам спрайта .SP1, а координаты окна внутри спрайта .SP2 определяются переменными .SCL и .SRW. Необходимо следить, чтобы окно в спрайте не «вылезло» за его край, иначе появится сообщение Parameter error.

Действие операторов этой группы не затрагивает атрибуты. Для перемещения атрибутов в этом случае используются операторы **.GMAT** и **.PMAT**.

Следует заметить, что при работе инструкций наложения «спрайт—окно спрайта» возникает побочный эффект: после выполнения любой из них запрещается перенос атрибутов для операций типа «спрайт—экран» и «экран—спрайт» (то есть автоматически выполняется .ATOF). Поэтому каждый раз нужно вновь разрешать перенос атрибутов оператором .ATON.

---

* В данном случае изображение перемещается из спрайта **.SP1** в спрайт **.SP2**. Но не для всех групп операторов это так. Чтобы предупредить возможную путаницу при использовании переменных .SP1 и .SP2, в Приложении 2 дана поясняющая таблица (табл. 13).

** Есть одна особенность в работе оператора **.PMXR**, происходящая, видимо, из-за ошибки в Laser Basic. После его использования в программе оператор **.PTOR** первый раз интерпретируется как **.PTXR**. Обойти эту ошибку можно, например, дважды выполнив **.PTOR**.

Перемещение спрайтов (5-й способ)

Развивая далее принцип перемещения спрайтов с сохранением фона в буфере (см. стр. 166), можно реализовать движение спрайта с шагом меньше знакоместа. Эта задача решается с помощью скроллинга спрайта в памяти (шаг скроллинга может быть равным 1, 4 и 8 пикселей) с выводом спрайта на каждом шаге в одно и то же окно экрана. При создании спрайта следует предусмотреть пространство для скроллинга — ободок шириной 8 пикселей (см. рис. 15).

Для примера рассмотрим, как можно реализовать перемещение спрайта с номером .SPN=1 слева направо на 10 знакомест с шагом в 1 пиксель и сохранением фона*:

```
10 .ROW=0:.LEN=4:.HGT=2: REM Окно в размер спрайта номер 1
20 .COL=5:.SPN=3:.PTBL: REM Фоном служит спрайт номер 3
30 FOR n=0 TO 10:.COL=n
40 .SPN=2:.GTBL: REM Помещение окна с фоном в буфер
50 .SPN=1:.PTOR: REM Вывод спрайта на экран
52 FOR m=1 TO 7:.WR1M: REM Скроллинг спрайта в памяти
54 .SPN=2:.PTBL: REM Восстановление окна из буфера
55 .SPN=1:.PTOR: REM Печать сдвинутого спрайта
56 PAUSE 10: NEXT m
58 .WR1M:.WL8M: REM Восстановление спрайта
60 PAUSE 10
70 .SPN=2:.PTBL: REM Восстановление окна из буфера
80 NEXT n
```

Скроллинг пейзажа

Перемещение на заднем плане какого-нибудь фонового рисунка (пейзажа) настолько часто встречается в играх, что есть смысл рассмотреть отдельно способы реализации этого приема. Проще всего организовать скроллинг протяженного пейзажа, прокручивая спрайт длиной в несколько экранов**.

Если шаг скроллинга — знакоместо (или несколько знакомест), то особых проблем не возникает. Такой вид скроллинга позволяет прокручивать изображение с атрибутами, хотя и не обеспечивает плавности движения. Для примера рассмотрим перемещение пейзажа высотой 3 строки и длиной 96 знакомест. Допустим, что спрайт с изображением пейзажа имеет номер 1 (отметим, что в спрайт-файле SPRITE2B такой спрайт отсутствует):

```
10 .SRW=0:.HGT=3:.ROW=10:.SPN=1: REM Начальные параметры
20 .COL=0:.PTBL: REM Вывод первых 32 столбцов пейзажа
```

* Правда, пример дает лишь принципиальное представление о данном способе, поскольку у выбранного спрайта, как и у всех других в спрайт-файле SPRITE2B, отсутствует ободок.

** О том, как его создать, см. в разделе «Создание спрайтов» на стр. 185.

```
30 FOR n=32 TO 96
40 .COL=0:.LEN=32:.SL8V:.ATLV: REM Скроллинг окна влево
50 .COL=31:.LEN=1:.SCL=n:.PWBL: REM Вывод n-го столбца спрайта
   в 31-й столбец экрана
60 NEXT n
```

Чтобы обеспечить перемещение пейзажа с шагом меньше знакоместа, придется применить небольшую хитрость. Нужно сделать 31-й столбец невидимым: задать в нем одинаковый цвет фона (PAPER) и тона (INK). Добавим к предыдущему примеру 15-ю строку и изменим 40-ю и 50-ю:

```
15 .COL=31:.LEN=1: PAPER 7: INK 7:.SETV: REM Обесцвечивание
   31-го столбца
40 FOR m=1 TO 8: .COL=0:.LEN=32:.SL1V: NEXT m: REM
   Скроллинг с шагом в 1 пиксель
50 .COL=31:.LEN=1:.SCL=n:.ATOF:.PWBL:.ATON: REM Вывод n-го
   столбца спрайта в 31-й столбец экрана
```

Если скроллинг идет вправо, то «скрытое» окно организуется у левого края экрана.

Изменение размеров области спрайт-файла

Для того чтобы было понятно дальнейшее изложение, скажем несколько слов о структуре спрайт-файла и его расположении в памяти (см. также распределение памяти интерпретатора Laser Basic: табл. 10 на стр. 178). Интерпретатор Laser Basic определяет, в каком месте находится спрайт-файл, по содержимому специальных ячеек памяти. Ячейки 62464/65 содержат адрес нижней границы спрайт-файла (в дальнейшем мы будем называть его START), а 62466/67 — адрес верхней (END). Следовательно, адреса границ спрайт-файла можно определить так:

```
LET START=PEEK 62464+256*PEEK 62465: PRINT START
LET END=PEEK 62466+256*PEEK 62467: PRINT END
```

Верхняя граница (END) обычно выбирается равной 56575 (и лучше всего ее не менять). В ячейке с адресом END всегда должен быть записан ноль — метка конца спрайт-файла. В противном случае могут возникнуть неприятные сюрпризы. Нижняя граница (START) — плавающая, она зависит от размера спрайт-файла.

Объем памяти (в байтах), занимаемый одним спрайтом, рассчитывается по формуле:

9×.HGT×.LEN+5

– где .HGT и .LEN — высота и ширина спрайта в знакоместах.

Формат хранения спрайта в спрайт-файле приведен в табл. 8 (адреса увеличиваются снизу вверх).

Таблица 8. **Формат хранения спрайта в памяти.**

данные об атрибутах спрайта	.HGT×.LEN байт
данные о состоянии пикселей спрайта	8×.HGT×.LEN байт
высота спрайта (.HGT)	1 байт
ширина спрайта (.LEN)	1 байт
адрес следующего спрайта	2 байта
номер спрайта	1 байт

Laser Basic позволяет в процессе работы программы расширять область спрайт-файла: резервировать место для нового спрайта и, наоборот, «выбрасывать» спрайты из этой области.

Чтобы расширить область спрайт-файла для нового спрайта, нужно после задания номера спрайта, его высоты и ширины (переменные .SPN, .HGT и .LEN) выполнить один из операторов: **.ISPR** или **.SPRT**. Они отличаются друг от друга тем, что .ISPR расширяет область спрайт-файла за счет понижения его нижней границы, а .SPRT — повышения верхней. Обратим внимание на то, что этими операторами только резервируется место под новый спрайт, а само изображение нужно записывать в спрайт-файл с помощью оператора .GTBL.

При создании спрайта с уже существующим номером с помощью оператора .SPRT старый спрайт уничтожается и замещается новым. Оператор же .ISPR в этом случае просто не выполняется. Поэтому перед использованием .ISPR нужно убедиться в отсутствии спрайта с заданным номером (о том, как это сделать, будет сказано позже). Если же спрайт существует, то его следует предварительно уничтожить оператором **.DSPR** или **.WSPR** (о них будет рассказано в этом разделе).

Расширение области спрайт-файла требует особой осторожности. Необходимо следить, чтобы нижняя граница спрайт-файла (START) не пересекла RAMTOP и не испортила бейсик-программу, а верхняя (END) не затерла коды интерпретатора, начинающиеся с адреса 56576. Напомним, что адрес RAMTOP можно определить так:

```
LET RAMTOP=PEEK 23730+256*PEEK 23731: PRINT RAMTOP
```

Приведем пример создания спрайта в спрайт-файле:

```
10 .SPN=240:.PTBL: REM Такого спрайта еще нет
20 CLEAR 50289: REM Опустим RAMTOP
30 .HGT=1:.LEN=1:.SPN=240:.ISPR: REM Создадим его
40 .SPN=240:.PTBL: REM И выведем на экран
```

При выполнении 10-й строки компьютер выдаст сообщение об ошибке Parameter error (ошибочный параметр) и будет прав: спрайта с номером 240 в спрайт-файле SPRITE2B нет. Выполнив GO TO 20,

убедимся, что все в порядке — спрайт создан, хотя информации в нем нет (точнее, она случайная).

Чтобы удалить спрайт номер .SPN из области спрайт-файла, существуют операторы .DSPR и .WSPR. Один, соответственно, сдвигает нижнюю границу спрайт-файла, другой — верхнюю.

Иногда может возникнуть необходимость переместить спрайт-файл в памяти, к примеру, чтобы зарезервировать пространство между верхней границей области спрайтов и началом интерпретатора под какие-либо подпрограммы в кодах. Для этих целей служит оператор **.RLCT**. Перед его выполнением нужно присвоить значение специальной переменной .MLN, задающей смещение спрайт-файла (в байтах). Теоретически эта переменная может принимать значения из интервала от 0 до 65535. Реально же нужно следить, чтобы перемещаемый спрайт-файл не попал в «жизненно важные» области памяти. Если спрайты нужно переместить вверх, то необходимо .MLN приравнять величине смещения. Если вниз, то .MLN должна равняться 65536–<смещение>.

Вспомогательные графические операторы и функции

Операторы и функции данной группы сами не выполняют преобразований графических объектов, но имеют к этому непосредственное отношение.

Оператор **.ADJV** «обрезает» переменные .HGT и .LEN так, чтобы окно, заданное значениями .ROW, .COL, .HGT и .LEN, не выходило за границы экрана. Оператор .ADJV выполняет действия, эквивалентные работе следующей программки:

```
10 LET h=?HGT: LET l=?LEN: LET r=?ROW: LET c=?COL
20 IF r+h>24 THEN .HGT=24–r
30 IF c+l>32 THEN .LEN=32–l
```

Оператор **.ADJM** изменяет значения графических переменных .HGT, .LEN, .SCL и .SRW таким образом, чтобы спрайт (или окно спрайта) при переносе на экран не выходил за его пределы. К примеру, если выполнить строку

```
10 .SPN=1:.ROW=1:.COL=31:.ADJM
```

– то переменная .LEN примет значение 1, так как на экране может поместиться только одна колонка спрайта.

Функция **?SCV** проверяет знакоместо экрана, заданное переменными .ROW и .COL, на наличие изображения. Если в тестируемом знакоместе есть включенные пиксели, то после выполнения программной строки

```
LET S=?SCV
```

переменной S присваивается ненулевое значение. Если знакоместо пусто, функция возвращает ноль.

Функция **?SCM** таким же образом проверяет спрайт с номером .SPN. Понятно, что любой непустой (содержащий изображение)

спрайт имеет включенные пиксели и функция возвратит ненулевое значение, если же спрайт пустой — ноль.

С помощью функции **?TST** можно выяснить некоторые параметры спрайта. Для этого задается номер проверяемого спрайта (.SPN) и выполняется строка

```
LET W=?TST
```

Если спрайт с указанным номером существует, то переменной W присваивается значение адреса его расположения в памяти, а графическим переменным .HGT и .LEN — значения размеров спрайта. При отсутствии тестируемого спрайта переменной W присваивается нулевое значение, а в переменных .HGT и .LEN может оказаться все что угодно.

Определение столкновений спрайтов

Одна из часто встречающихся задач при создании динамических игр — это определение факта столкновения спрайтов. В Laser Basic для этого используются уже описанные функции ?SCV и ?SCM.

Сначала рассмотрим наиболее простую задачу: как определить столкновение спрайта не с конкретным объектом, а с любой картинкой-препятствием на экране:

```
  10 PRINT AT 10,20;"STOP": REM Создание препятствия на экране
 100 .ROW=10:.COL=0:.SPN=1:.PTBL: REM Вывод на экран спрайта
 110 FOR n=1 TO 25:.HGT=0:.LEN=1:.SP1=1:.SP2=1
 120 .MOVE: GO SUB 130: NEXT n: REM Перемещение спрайта
 125 STOP
 130 LET c=?COL: REM Сохранение значения переменной .COL
 140 .COL=c+4: LET crash=?SCV: REM Проверка знакоместа перед
     спрайтом на наличие включенных пикселей
 150 IF crash=0 THEN .COL=c: RETURN: REM Если препятствие не
     обнаружено, то .COL присваивается исходное значение
1000 BEEP 1,1: REM Столкновение
```

Сложнее определить столкновение спрайта с конкретным объектом. В следующем примере в качестве такого объекта используется спрайт .SPN=2. К программе, фиксирующей в предыдущем примере факт столкновения, добавлены строки 200...240. В них происходит сравнение объекта, с которым произошло столкновение, и спрайта .SPN=2. Спрайт .SPN=5 используется в качестве буфера. Его размеры должны быть равны размерам спрайта .SPN=2.

```
200 .SPN=5: LET T=?TST:.GWBL: REM Копирование препятствия в бу-
    фер
210 .SP1=5:.SP2=2
220 .PMXR: REM Наложение изображения заданного объекта на
    содержимое буфера по принципу XOR
230 LET crash=?SCM: REM Проверка буфера на наличие включен-
    ных пикселей
```

```
240 IF crash<>0 THEN .COL=c: RETURN: REM Если включенных
      пикселей нет (crash=0), значит заданный объект и препятствие
      совпадают
1000 BEEP 1,1: REM Столкновение с заданным объектом
```

В этом примере факт столкновения зафиксирован не будет, поскольку препятствие на экране не совпадает со спрайтом .SPN=2. Но если теперь заменить 10-ю строку на

```
10 .ROW=10:.COL=20:.SPN=2:.PTBL
```

— то столкновение произойдет.

В приведенных примерах предполагалось, что столкновение спрайтов происходит на чистом экране. Для решения усложненной задачи — определения столкновения спрайтов на неком фоне, перед проверкой совпадения спрайтов необходимо содержимое буфера очистить от фона, наложив на буфер спрайт .SP2=2 по принципу AND (например, оператором .PMND), а после этого уже выполнить оператор .PMXR в строке 220.

Сервисные операторы и функции

Инструкции, описанные в этом разделе, расширяют возможности Spectrum-Бейсика и не имеют никакого отношения к преобразованию графических объектов.

Функция **?KBF** используется для контроля нажатия конкретной клавиши. Как и все другие функции Laser Basic, она может использоваться только в строке типа

```
LET b=?KBF
```

?KBF возвращает ненулевое значение, если в момент выполнения оператора LET была нажата проверяемая клавиша.

Клавиша, на нажатие которой будет реагировать функция, задается переменными .ROW и .COL в соответствии с табл. 9.

Таблица 9. **Задание клавиши для опроса функцией ?KBF.**

.ROW	.COL				
	1	2	3	4	5
1	CS	Z	X	C	V
2	A	S	D	F	G
3	Q	W	E	R	T
4	1	2	3	4	5
5	0	9	8	7	6
6	P	O	I	U	Y
7	Enter	L	K	J	H
8	Break	SS	M	N	B

Приведем пример программки, ожидающей нажатия клавиши **K**:

```
10 .ROW=7:.COL=3: REM Указываем клавишу
20 LET s=?KBF: IF s=0 THEN GO TO 20: REM Цикл ожидания
30 BEEP 1,1: REM Дождались
```

Оператор **.POKE** addr,x является 16-битовой версией оператора Spectrum-Бейсика POKE*: он записывает двухбайтовое число x в последовательно расположенные ячейки памяти с адресами addr и addr+1, то есть эквивалентен инструкциям

```
POKE addr,INT(x/256): POKE(addr+1),x–INT(x/256)
```

Нетрудно догадаться, что функция **?PEK** — это 16-битовый аналог функции PEEK стандартного Бейсика. Она возвращает значение двухбайтового числа, записанного в двух ячейках памяти: адрес первой равен аргументу функции, адрес второй на единицу больше. Например, оператор

```
LET x=?PEK 23606
```

эквивалентен

```
LET x=PEEK 23606+256*PEEK 23607
```

Laser Basic позволяет осуществлять отладку (трассировку) программ: оператор **.TRON** включает пошаговое выполнение программы**. Во время трассировки на экран выводится текст выполняемой в данный момент строки. Эта особенность резко сужает сферу применения оператора .TRON, так как после выполнения нескольких операторов весь экран заполнится строчками листинга программы. Отключается трассировка с помощью оператора **.TROF**. Оба описываемых оператора могут находиться только в тексте программы, ввод их с клавиатуры не вызовет никакого действия.

Оператор **.RNUM** N , M, S перенумеровывает строки программы: строке N присваивается номер M, следующей строке M+S и т. д. (по умолчанию параметр S принимается равным 10). Так, например, .RNUM 1, 20, 15 перенумерует все строки программы с шагом 15, первой строке будет присвоен номер 20. При этом соответствующим образом заменяются параметры операторов GO TO, GO SUB и RESTORE, но только в том случае, если они представляют собой численные значения, а не выражения. Если использовать .RNUM без параметров, то у первой строки номер не меняется, а последующие перенумеруются с шагом 10.

Оператор **.REMK** для уменьшения объема памяти, занимаемой программой, удаляет из нее комментарии (все операторы REM и следующие за ними тексты). Необходимо быть осторожным при

* Как видно, формат .POKE не стандартен для Laser Basic — его параметры передаются не через графические переменные, а непосредственно вслед за ключевым словом.

** Пошаговый режим означает, что после выполнения каждого оператора ожидается нажатие любой клавиши для продолжения трассировки.

использовании этого оператора в программах, обращающихся к дисководу. Иначе вместо строчки

```
RANDOMIZE USR 15619: REM: LOAD "NAME" CODE
```

можно получить

```
RANDOMIZE USR 15619
```

Процедуры

Laser Basic не только расширяет графические возможности стандартного Spectrum-Бейсика, но и дополняет его новыми средствами структурного программирования: инструментом процедур.

Процедура — это, по сути дела, подпрограмма, имеющая свои собственные внутренние переменные, называемые *локальными*. Значения локальных переменных в процессе выполнения процедуры могут меняться, но при выходе из процедуры им возвращаются значения, которые они имели в основной программе. А главное, посредством локальных переменных в процедуру передаются необходимые параметры. Таким образом, процедура, в отличие от подпрограммы, позволяет создавать независимые структурные блоки, которые можно использовать в других программах, не боясь «конфликта переменных».

Процедуры отличаются от подпрограмм также способом их определения и вызова. В Laser Basic каждой процедуре присваивается имя: буква латинского алфавита со знаком $ или без него (всего возможно 52 имени). Определяются процедуры в любом месте программы оператором DEF FN с именем процедуры (набирается DEF FN способом, принятым для Spectrum-Бейсика, клавишами CS/SS+SS/1)*. А чтобы различить процедуру и функцию, определяемую пользователем, после имени процедуры ставится знак #. Например: DEF FN A# или DEF FN q$#.

Блок программы, оформляемый как процедура, начинается с так называемого *заголовка процедуры*, который может выглядеть, например, так:

```
100 DEF FN A#(x,y,z,a$,b$)
```

В заголовке после имени процедуры в скобках** приводится перечень *формальных параметров*, то есть числовых и строковых переменных, с которыми надлежит работать процедуре. Формальными параметрами не могут быть массивы.

В процедуре могут использоваться и переменные, не указанные в списке. Такие переменные называются *глобальными*. При выходе из процедуры им не будет возвращено значение, которое они имели при входе в нее.

---

* Отметим, что выполнение процедур, как и функций, заданных пользователем, происходит тем быстрее, чем ближе к началу программы расположено их определение.

** Скобки в описании процедуры должны стоять даже при отсутствии передаваемых параметров.

Текст процедуры должен оканчиваться оператором .RETN (аналог RETURN).

Процедуры вызываются оператором .PROCFN с указанием вслед за ним имени процедуры и, в скобках, списка *фактических параметров*, то есть значений всех переменных, упомянутых в заголовке процедуры. Набирается этот оператор необычным способом: первая его часть (.PROC) — по буквам, а FN — как в Spectrum-Бейсике (CS/SS+SS/2).

Описание процедуры может включать в себя и вызов другой процедуры, но при этом надо помнить, что значения локальных переменных для внутреннего вызова распространяются и на внешний вызов, то есть эти переменные «не до конца локальны».

Рассмотрим несколько примеров. Вот так будет выглядеть определение процедуры, выводящей на экран слово HELP!:

```
100 DEF FN a$#()
200 PRINT "HELP!"
300 .RETN
```

Вызывается она просто:

```
50 .PROCFN a$#()
```

Можно сделать так, чтобы процедура выводила любой текст в заданном месте:

```
100 DEF FN a$#(x,y,b$)
200 PRINT AT x,y;b$
300 .RETN
```

После определения этой процедуры для печати, например, слова LASER с 8-й позиции 5-й строки, в программу достаточно вписать:

```
50 .PROCFN a$#(5,8,"LASER")
```

Загрузка и запись программ

Программы, написанные в интерпретаторе Laser Basic, могут быть загружены и записаны как в непосредственном режиме, так и из программы. Причем, если запись производилась из программы, то и загрузка должна происходить из программы. И, наоборот, программы, записанные в непосредственном режиме, могут быть загружены только с клавиатуры.

Запись программ без автостарта осуществляется операторами:

SAVE "NAME" (для ленты),
RANDOMIZE USR 15619: REM: SAVE "NAME" (для диска).

Запись программы с автостартом со строки N осуществляется операторами:

SAVE "NAME" LINE N (для ленты),
RANDOMIZE USR 15619: REM: SAVE "NAME" LINE N (для диска).

При запуске записанная таким образом программа должна до встречи какого-либо оператора Laser Basic выполнять один из перечисленных операторов:

RUN, GO TO или GO SUB (для ленты)
RANDOMIZE USR 58841 и RUN, GO TO или GO SUB (для дисковода)

Ниже приведен пример загрузчика, который подгружает интерпретатор Laser Basic, программу и спрайт-файл:

```
10 CLEAR START—1: REM Установка RAMTOP на 1 байт меньше,
   чем нижняя граница спрайт-файла
20 LOAD "LASOBJ" CODE: REM Загрузка кодов интерпретатора
30 LOAD "LASLOBJ" CODE
40 RANDOMIZE USR 62464
50 LOAD "GRAPH" CODE
60 RANDOMIZE USR 58841: REM Запуск интерпретатора
70 LOAD "имя" CODE START: REM Загрузка спрайт-файла
80 GO TO 90: REM Выполнение оператора перехода (необходимо
   перед первой командой Laser Basic)
90 .POKE 62464,START:.POKE 62466,56575: REM Заполнение ячеек,
   хранящих адреса START и END
100 POKE 56575,0
```

Если в программе не используется спрайт-файл, то выбрасываются строки 70 и 90, а 10-я заменяется на

```
10 CLEAR 56574
```

В заключение описания интерпретатора Laser Basic представим распределение памяти при его использовании (табл. 10).

Таблица 10. **Распределение памяти при работе с интерпретатором.**

P_RAMT (23732)		65535
	Интерпретатор Laser Basic	
END (62466/67)		56575
	Область спрайт-файла	
START (62466/67)		
RAMTOP=START-1 (23730)		
	...	
E_LINE (23641)		
	Переменные Бейсика	
VARS (23627)		
	Бейсик-программа	
PROG (23635)		

СОЗДАНИЕ СПРАЙТОВ

В этом разделе мы расскажем о двух программах — генераторах спрайтов: **Sprite Generator (SPTGEN)** из фирменного пакета Laser Basic и **Spriter***, написанной автором этой главы. Обе программы предназначены для создания спрайт-файла — кодового файла, содержащего набор спрайтов, готовых для использования в Laser Basic.

Основное различие описываемых генераторов спрайтов в том, что фирменная программа SPTGEN ориентирована на создание изображения «с нуля» по точкам, а Spriter позволяет использовать для этого готовые картинки, нарисованные в любом графическом редакторе для ZX Spectrum (Art Studio, Artist II и другие) или даже позаимствованные из фирменных «игрушек». Как нам кажется, в этом смысле Spriter удобнее, так как специальные графические редакторы обладают гораздо большими возможностями для формирования картинок, чем SPTGEN. Кроме того, SPTGEN позволяет создавать спрайты с максимальным размером только 15×15 знакомест, а Spriter — 32×22, то есть почти во весь экран. Однако у каждого свой вкус, поэтому мы подробно опишем обе программы.

Программа SPTGEN

Фирменный генератор спрайтов, по сути, является примитивным графическим редактором. Процесс создания спрайт-файла с помощью SPTGEN выглядит так:

1. оперируя клавиатурой, пользователь создает рисунок одного знакоместа;
2. рисунок знакоместа помещается в спрайт;
3. построенный из отдельных знакомест спрайт записывается в память;
4. несколько спрайтов сохраняются в виде спрайт-файла на диске или ленте.

Загружается спрайт-генератор оператором LOAD "SPTGEN". В нижней строке экрана появляется запрос: COLD OR WARM START? («холодный» или «теплый» старт?). При «холодном» старте программа запускается «с нуля», то есть очищается область спрайт-файла. «Теплый» старт используют для продолжения работы после случайного или намеренного выхода из SPTGEN в Бейсик. В этом случае нужно выполнить оператор GO TO 2 и выбрать «теплый» старт, нажав клавишу **W**. Таким образом можно сохранить в памяти спрайт-файл (но не экранную картинку, ее придется рисовать заново).

---

* Программа Spriter написана на языке Laser Basic, и сама по себе может служить примером использования некоторых его возможностей.

Сразу после загрузки программы нужно осуществить «холодный» старт, то есть нажать клавишу **C**. Затем на новый запрос: COLD START (Y/N), чтобы подвердить «температуру старта», придется нажать клавишу Y. Программа SPTGEN вообще очень недоверчива и постоянно требует от пользователя подтверждения его действий. Наконец, после окончания диалога, на экране появится картинка (рис. 17).

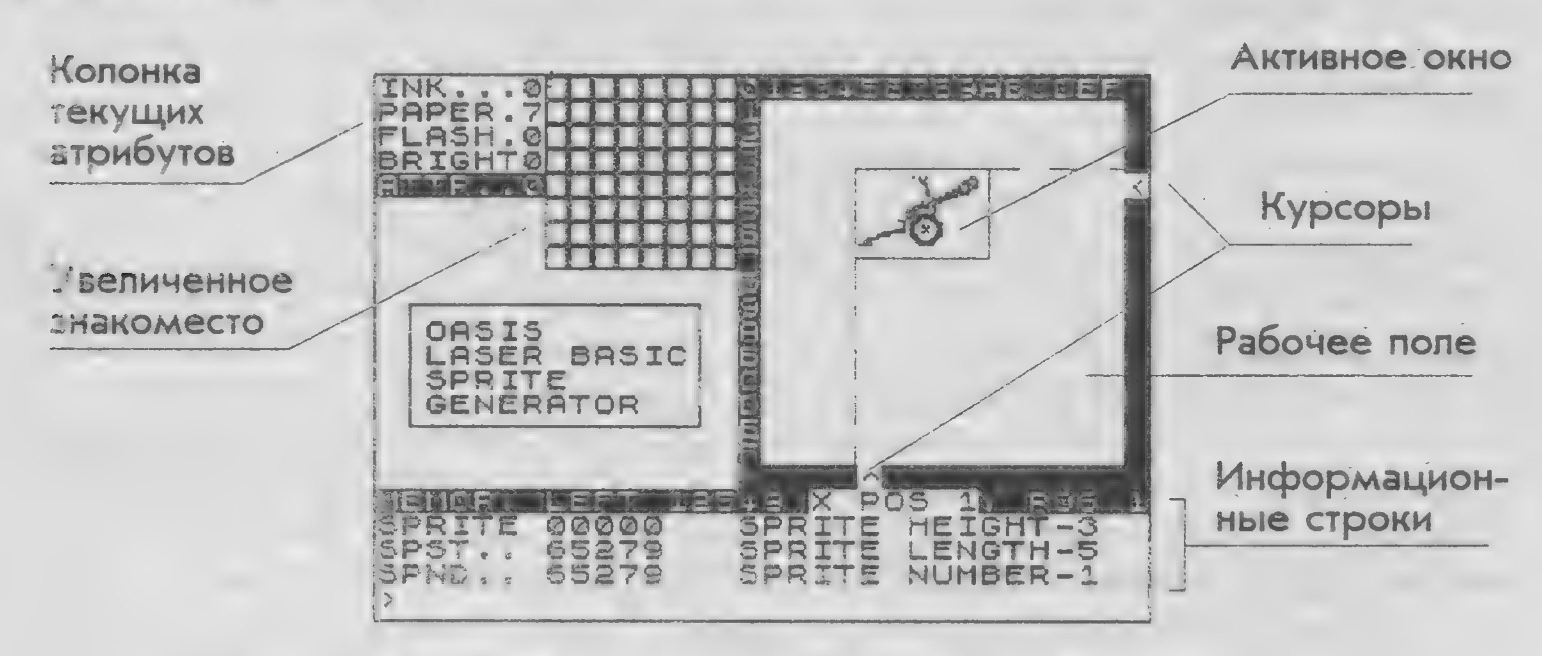


Рис. 17. Рабочий экран генератора спрайтов SPTGEN.

Большой квадрат размером 15×15 знакомест в правом верхнем углу — это *рабочее поле*, в котором формируется спрайт. Спрайт с заданным номером будем называть *текущим спрайтом*. В правом верхнем и левом нижнем углах рабочего поля мигают два *курсора*: вертикальный и горизонтальный. Знакоместо, расположенное на пересечении этих курсоров, назовем *активным знакоместом*. Его увеличенное изображение, именуемое *увеличенным знакоместом*, находится левее рабочего поля. В рабочем поле выделяют *активное окно*, верхний левый угол которого совпадает с активным знакоместом, а ширина и высота задаются соответствующими командами. Еще левее — столбец текущих атрибутов. Внизу размещены информационные строки:

MEMORY LEFT	– свободный объем памяти для размещения спрайтов (в байтах);
X POS, Y POS	– горизонтальная и вертикальная координаты курсора рабочего поля (1...15);
SPRITE	– начальный адрес размещения в памяти текущего спрайта;
SPRITE HIGHT	– высота активного окна (в знакоместах);
SPRITE LENGHT	– ширина активного окна (в знакоместах);
SPST	– начальный адрес спрайт-файла в памяти (сразу после загрузки — 65279);
SPND	– конечный адрес спрайт-файла;
SPRITE NUMBER	– номер текущего спрайта (1...255).

Теперь более подробно расскажем о том, как работать с программой SPTGEN.

Создание и редактирование изображения в увеличенном знакоместе. Перед началом создания новых спрайтов желательно подготовить на разграфленном листе бумаги «проекты» их изображений. Это намного упростит и ускорит работу. Ведь в SPTGEN спрайты строятся из отдельных кирпичиков — знакомест, и удержать в голове весь «проект» (то есть как будет выглядеть спрайт целиком) достаточно сложно. Прорисовка знакомест идет в увеличенном знакоместе: клавишами **5**, **6**, **7**, **8** выбирается требуемый пиксель, клавишами **9** и **0** он, соответственно, включается и выключается.

Перенос изображения на рабочее поле. Копирование увеличенного знакоместа в активное знакоместо осуществляется по команде (клавише) **J**. Позиция активного знакоместа устанавливается клавишами **SS/5**, **SS/6**, **SS/7**, **SS/8**. По команде **K** ранее созданное знакоместо возвращается на редактирование, либо по команде **Q** стирается. При желании можно очистить и все рабочее поле (**SS/Q**).

Метод прямого ввода данных. Создать элемент изображения спрайта можно, и не прибегая к построениям в увеличенном знакоместе. После нажатия клавиши **D** программа SPTGEN позволяет последовательно, по нажатию **Enter**, ввести восемь значений: восемь байт данных, соответствующих одному знакоместу (как при построении символов, определяемых пользователем). Значения, естественно, должны лежать в пределах 0...255 (десятичные) или H00...HFF (шестнадцатеричные). Изображение побайтно переносится в активное знакоместо рабочего поля.

Управление атрибутами. При переносе изображения из увеличенного знакоместа на рабочее поле и обратно ему (изображению) могут присваиваться текущие атрибуты, а могут и не присваиваться — все зависит от того, какой режим был задан: нажатие клавиш **A**, затем **1** (то есть **A+1**) разрешает перенос атрибутов, **A+0** — запрещает.

Значения текущих атрибутов устанавливаются клавишами **X** (цвет тона: 0...7), **C** (цвет фона: 0...7), **B** (яркость: 0, 1) и **V** (мерцание: 0, 1). Кроме такого последовательного способа определения текущих атрибутов, в процессе работы их можно изменить все одновременно, присвоив им значения атрибутов активного знакоместа (клавиша **U**). Возможна и обратная операция: заполнение активного знакоместа текущими атрибутами (клавиша **I**).

Операции в активном окне. В процессе создания спрайта его изображение на рабочем поле можно трансформировать. Операции над изображением осуществляются в пределах активного окна, верхний левый угол которого определяется положением активного знакоместа, то есть курсорами рабочего поля, а высота и ширина задаются, соответственно, командами **H** (1...255) и **L** (1...255). Границы активного окна постоянно на экране не отображаются, но их всегда можно увидеть, удерживая клавишу **F**, — изображение в пределах окна начинает мерцать.

Из всевозможных операций над изображением в SPTGEN реализованы лишь самые элементарные: циклический скроллинг вправо, влево, вверх и вниз (команды, соответственно, **SS/I**, **SS/T**, **SS/U** и **SS/Y**), инверсия (клавиши **E+1**), зеркальное отображение

картинки (**E+2**) и атрибутов (**E+3**) относительно вертикальной оси окна.

Запись спрайта. Спрайт с текущим номером записывается в память (в спрайт-файл) нажатием клавиши **G**. Размеры его определяются границами активного окна. Перед записью необходимо проверить правильность совмещения окна с изображением (клавиша **F**).

Изменение номера текущего спрайта осуществляется по команде **S** (1...255). Если указывается уже существующий номер, будет выдано сообщение о совпадении номеров.

При записи по команде **G** ранее существующий в спрайт-файле спрайт с текущим номером заменяется новым. При необходимости изображение старого и созданного спрайтов можно совместить. Делается это с помощью команды **O**, реализующей три принципа наложения: OR, XOR и AND (см. стр. 164), которые задаются, соответственно, нажатием одной из клавиш: **1**, **2** или **3**, после **O**.

Изображение с рабочего поля может переноситься в память как с текущими атрибутами, так и без них, в зависимости от того, разрешен или запрещен перенос текущих атрибутов (команда **A+1** или **A+0**).

Проверить правильность записи спрайта в память или вызвать его на редактирование можно клавишей **P**. По этой команде текущий спрайт из памяти копируется в активное окно.

С помощью нажатия клавиши **Space** можно наложить изображение активного окна на спрайт с текущим номером. Конечно, это возможно, только если размеры окна меньше размеров спрайта в памяти. Команда запрашивает координаты знакоместа в спрайте, куда будет наложено изображение (INPUT COLUMN, INPUT ROW). Принцип наложения задается нажатием одной из клавиш:

1 – окно перемещается в спрайт, затирая фон;
2 – наложение по принципу OR;
3 – наложение по принципу XOR;
4 – наложение по принципу AND.

При необходимости можно создать и пустой спрайт с шириной и высотой, заданными пользователем в интервале от 1 до 255 знакомест (команда **SS/C**). Размеры спрайта вводятся на запросы INPUT LENGHT, INPUT HIGHT.

Преобразование спрайтов в памяти. SPTGEN позволяет трансформировать спрайты не только на экране, но и в памяти. Команда **M** выполняет тот же минимум преобразований с текущим спрайтом в памяти, что и команда **E** с изображением в активном окне: инверсия (**M+1**), зеркальное отображение изображения (**M+2**) и атрибутов (**M+3**) относительно вертикальной оси окна.

Кроме того, программа предоставляет возможность скопировать текущий спрайт, повернутый на 90°, в спрайт с другим номером — команда **R**. Номер второго спрайта вводится на запрос программы.

По команде **W** происходит удаление из спрайт-файла текущего спрайта. При этом все спрайты, находящиеся в памяти ниже изъятого, поднимаются вверх, занимая свободное пространство.

Команда тестирования спрайтов **T** (аналогичная функции ?TST) выводит на экран параметры текущего спрайта: в графы SPRITE

HIGHT и SPRITE LENGTH — его высоту и ширину, в графу SPRITE — адрес его размещения в памяти.

Запись и загрузка спрайт-файла. После создания спрайт-файла можно записать его на ленту или диск, нажав клавиши **SS/S**. Программа выдаст запрос о формате записи (OPTION1 — OPTION2), в ответ на который требуется нажать клавишу **1** или **2**:

1 – спрайт-файл записывается в формате, допускающем последующую загрузку файла в SPTGEN и редактирование, но исключающем возможность использования его в программах на Laser Basic;

2 – спрайт-файл записывается в формате для использования в программах, но его уже «не возьмет» генератор спрайтов.

После задания формата записи спрайт-файла на соответствующий запрос вводится его имя. Нужно нажать одну из пяти клавиш: **1**, **2**, **3**, **4**, и **5**. Соответственно, спрайт-файл запишется под именем "1", "2" и т. д.

Перед записью в информационной строке появится сообщение о нижней границе спрайт-файла:

FILE = N ADDRESS = XXXXX

– где N — имя (номер) спрайт-файла, а XXXXX — его нижняя граница (START). Это значение следует запомнить: именно его необходимо вводить при загрузке спрайт-файла с помощью программы-диспетчера Laser.

После записи спрайт-файла автоматически выполняется оператор сравнения VERIFY. Если при сравнении произойдет сбой, программа «вываливается» в Бейсик. Для возврата в SPTGEN с сохранением спрайт-файла нужно выполнить оператор GO TO 100.

Загружаются спрайт-файлы с ленты или диска по команде **SS/J**. После нажатия клавиши программа выдаст серию запросов:

LOAD SPRITES FROM TAPE (Y/N) (загрузить с ленты?)
LOAD SPRITES FROM DISK (Y/N) (загрузить с диска?)
INPUT SPRITE FILE (1–5) (введите имя спрайт-файла)

После загрузки спрайт-файла находящиеся в памяти спрайты, естественно, стираются.

Программа Spriter*

Генератор спрайтов **Spriter** позволяет создавать спрайты размером до 32×22 знакомест и формировать спрайт-файл длиной до 26 килобайт. Готовые спрайт-файлы и экранные файлы загружаются в компьютер и записываются на ленту или диск непосредственно в процессе работы. В программе используется один формат записи спрайт-файла, допускающий и последующее редактирование файла в генераторе, и использование его в программах на Laser Basic.

---

* Текст программы приведен в Приложении

Алгоритм создания спрайт-файла программой Spriter. Картинки спрайтов рисуются в любом графическом редакторе*. Затем экранный файл со спрайтами записывается на ленту или диск. Естественно, можно пользоваться и готовыми экранными файлами.

С помощью программы Spriter спрайты «снимаются» с экрана и помещаются в спрайт-файл, который затем может быть записан на ленту или диск.

Программа загружается оператором LOAD "SPRITER". Сразу после загрузки появится запрос о типе магнитного накопителя: Tape or Disk (T/D). Для работы с магнитофоном нажимается клавиша **T**, с дисководом — **D**.

Далее, на запрос: Screen name?, вводится имя экранного файла с изображениями спрайтов.

После загрузки экранного файла в двух нижних строках появляется информация о текущих параметрах, которая будет постоянно находиться перед глазами:

.SPN – номер текущего спрайта;
.HGT – высота активного окна (в знакоместах);
.LEN – ширина активного окна (в знакоместах);
START – адрес начала спрайт-файла (нижняя граница);
LENGHT – длина спрайт-файла.

В верхнем левом углу экрана размещается активное окно. Оно задает границы спрайтов, переносимых в память. Размеры окна — ширина и высота — легко изменяются следующими клавишами:

CS/5 – уменьшение ширины активного окна на знакоместо;
CS/6 – увеличение высоты активного окна на знакоместо;
CS/7 – уменьшение высоты активного окна на знакоместо;
CS/8 – увеличение ширины активного окна на знакоместо.

Клавишами **5**, **6**, **7** и **8** весь экран циклически прокручивается по четырем направлениям с шагом в одно знакоместо. Таким образом, любой фрагмент экрана может быть помещен в активное окно, а оттуда перенесен в память в виде спрайта. При этом, соответственно, изменится содержание информационных строк. При изменении размеров активного окна и выполнении некоторых других операций размеры окна индицируются инверсией.

Управляющие клавиши программы Spriter:

N – ввод нового номера текущего спрайта (.SPN). Допускаются значения от 1 до 255;

G – создание спрайта с номером .SPN — копирование изображения из активного окна в спрайт-файл;

P – перенос спрайта с номером .SPN из памяти в активное окно;

D – стирание (удаление из спрайт-файла) спрайта с номером .SPN;

C – загрузка нового экранного файла с ленты или диска. На запрос Screen name? вводится имя экранного файла;

---

* При создании экранов помните, что нижние две строки будут заняты генератором спрайтов для вывода служебной информации.

- **S** – запись готового спрайт-файла на диск или ленту. Перед записью следует запомнить значение переменной START, индицируемое в информационной строке (нижняя граница спрайт-файла). Оно потребуется при загрузке спрайт-файла с помощью программы-диспетчера Laser;
- **L** – загрузка спрайт-файла для редактирования. Перед загрузкой в ответ на запрос Starting address? следует ввести адрес нижней границы загружаемого спрайт-файла. При загрузке нового спрайт-файла спрайты, ранее занесенные в память, будут утрачены;
- **V** – показ границ активного окна;
- **B** – выход в интерпретатор Laser Basic. Команда может понадобиться для выполнения других инструкций Laser Basic, не предусмотренных генератором спрайтов.

Любой запрос может быть отменен, если в ответ на него просто нажать Enter. Исключение составляет запрос Starting address?, который отменяется вводом недопустимого значения стартового адреса (меньше 30000 или больше 56575).

В процессе работы в нижней строке экрана будут появляться сообщения, понятные любому программисту, так как в них используется не более десятка английских слов.

Если при работе Вы случайно (или нарочно) «вылетите» в Бейсик, то вернуться в программу можно, дав команду GO TO 0 или RUN. При этом спрайт-файл сохраняется, но картинка на экране не восстановится.

Создание спрайтов больших размеров

Оба рассмотренных выше генератора спрайтов накладывают ограничения на максимальный размер создаваемого спрайта. Для программы Spriter этот предел — 32×22 знакоместа, для SPTGEN и того меньше — 15×15. Однако спрайт любых размеров, в том числе и спрайт длиной в несколько экранов, можно легко добавить в готовый спрайт-файл, не пользуясь генератором спрайтов. Для этого необходимо всего лишь выполнить несколько операторов Laser Basic.

Сначала рассмотрим алгоритм включения в готовый спрайт-файл нового спрайта с размером, меньшим или равным экрану:

1. создать изображение спрайта в виде экранного файла (в графическом редакторе) и сохранить его на диске или ленте. Для удобства изображение лучше размещать в левом верхнем углу экрана (чтобы не запоминать его положения);
2. загрузить интерпретатор Laser Basic;
3. выполнить CLEAR 25000;
4. загрузить готовый спрайт-файл с адреса, равного его нижней границе (назовем ее STARTOLD);
5. записать в ячейки 62464 и 62465 адрес нижней границы STARTOLD;

6. задать номер и размеры добавляемого спрайта в знакоместах, выполнив операторы .SPN=N:.LEN=L:.HGT=H;
7. выполнить следующие несколько строк:

```
10 LOAD "SCREEN"CODE 16384,6912: REM Загрузка экрана
20 .ISPR: REM Увеличение размеров спрайт-файла
30 .GTBL: REM Запись спрайта
```

8. определить новую нижнюю границу спрайт-файла:
   LET START=?PEK 62464
9. сохранить модифицированный спрайт-файл:
   SAVE "SPRITES"CODE START,56575–START
   – где START — новая нижняя граница спрайт-файла.

Описанным выше методом можно создать спрайт размером не больше экрана — 32×24 знакоместа. Преодолеть это ограничение позволяет другой метод. Рассмотрим его на примере создания спрайта длиной в 96 и высотой в 3 знакоместа в готовый спрайт-файл (использование такого спрайта описано в разделе «Скроллинг пейзажа» на стр. 169).

Предварительно нужно нарисовать спрайт в графическом редакторе, разместив его на экране в три полосы высотой по 3 знакоместа и длиной в 32 столбца. Первая полоса должна начинаться с верхней строки экрана, между полосами не должно быть пробелов.

Далее необходимо выполнить последовательность действий, описанную выше, дополнив программку из пункта 7 следующими строками:

```
30 .LEN=31:.HGT=3:.COL=0
40 .ROW=0:.SCL=0:.GWBL: REM Перенос в спрайт 1-й полосы
50 .ROW=3:.SCL=31:.GWBL: REM Перенос 2-й полосы
60 .ROW=6:.SCL=62:.GWBL: REM Перенос 3-й полосы
```

КОМПИЛЯТОР

Компилятор Laser Basic преобразует исходную бейсик-программу в некую конструкцию из кодов с целью сделать программу независимой от интерпретатора и повысить скорость ее работы.

Однако, хотя программа после компиляции и освобождается от опеки интерпретатора, она попадает в другую кабалу: не может работать без присутствия в памяти пакета рабочих процедур компилятора. Но обычно заботиться о размере программы не приходится: любая программа, написанная в интерпретаторе Laser Basic, сможет быть обработана компилятором*.

* Компилятор Laser Basic может транслировать и стандартные бейсик-программы, но без особого выигрыша в скорости

Ограничения на текст исходной программы

Основное, на что стоит обратить внимание при работе с компилятором, — это ограничения, накладываемые им на использование в компилируемой программе операторов Spectrum-Бейсика:

- запрещается использование операторов CLEAR, MERGE, CONTINUE, LIST, LLIST;
- операторы LOAD, SAVE, VERIFY допускаются только при условии, что они загружают или записывают на ленту откомпилированную программу (то есть они не могут работать, например, с некомпилированной программой, с кодовыми блоками). Команды обращения к диску в системе TR-DOS не выполняются вообще;
- в операторах RUN, GO TO, GO SUB, RESTORE допускаются только целочисленные параметры (нельзя использовать переменные и выражения);
- оператор RUN работает как обычно, но не производит очистки переменных;
- в операторе INPUT не допускается использовать ключевое слово LINE;
- операторы OPEN# и CLOSE# работают только с потоками "S" и "P";
- если в цикле FOR...NEXT содержится оператор Laser Basic, то конечное значение параметра цикла не должно быть меньше начального при положительном значении STEP или превосходить его при отрицательном STEP, то есть цикл должен быть выполнен хотя бы один раз. Пример:

  ```
  10 LET s=RND*10
  20 FOR n=5 TO s:.INVV: NEXT n
  ```

  Во избежание ошибок в указанный фрагмент следует вставить одну дополнительную строку:

  ```
  15 IF s<5 THEN GO TO 10
  ```

  Текст исходной программы сохраняется на магнитном носителе оператором SAVE "NAME".

Обратите внимание, что для нормальной работы скомпилированной программы исходная бейсик-программа должна стартовать с первой строки.

Компиляция

При компиляции в памяти компьютера должны находиться только текст исходной программы и сам компилятор.

Прежде всего нужно «сбросить» компьютер, выполнить CLEAR 59799 и оператором LOAD "NAME" загрузить исходную программу. После этого оператором LOAD "COMPCODE" CODE загружается компилятор.

Компиляция запускается оператором RANDOMIZE USR 59800.

Во время компиляции экранная область используется в качестве буфера, поэтому в верхних строках экрана отображается случайная информация. По окончании компиляции экран очистится и внизу появится сообщение типа 0 OK, 1000:1, которое означает, что все нормально, компиляция закончилась на строке 1000. Остается только сохранить полученный файл оператором SAVE "NAMEcomp" LINE 1. Перед записью не следует выполнять оператор CLEAR.

Полученный файл не поддается редактированию ни средствами стандартного Бейсика, ни средствами Laser Basic, поэтому есть смысл сохранить исходную (некомпилированную) программу.

Если взглянуть на листинг откомпилированной программы, то мы увидим всего 2-3 строки. Первая

```
0 PRINT 0
```

содержит информацию для пакета рабочих процедур и не выполняется. Вторая строка

```
0 RANDOMIZE USR 59800
```

запускает откомпилированную программу.

Если в исходной программе имеются функции, определяемые пользователем, то их список помещается в третьей строке с номером 0. Все операторы DATA помещаются в четвертую строку, также с номером 0.

К сожалению, компиляция не всегда заканчивается сообщением 0 OK. Она может быть прервана с выдачей номера строки, в которой встретилась ошибка, и одного из следующих сообщений:

Illegal statement found

обнаружен один из недопустимых операторов: CLEAR, MERGE, LIST, LLIST или CONTINUE;

Procedure definition nesting error

найдено два описания процедуры подряд без оператора .RETN между ними;

RETN without DEF in error

найден оператор .RETN без соответствующего описания процедуры;

Procedure not found

обращение к неописанной процедуре.

Загрузка откомпилированной программы

Для загрузки и запуска откомпилированной программы необходима отдельная программа-загрузчик, написанная на Spectrum-Бейсике.

Стандартный, универсальный загрузчик содержится в файле "LOADER":

```
10 CLEAR 59799:REM Пространство под рабочие процедуры*
20 LET spst=XXXXX: REM Нижняя граница спрайт-файла (START)
30 LOAD "RTCODE" CODE: REM Загрузка рабочих процедур
40 LOAD "SPRITES" CODE spst: REM Загрузка спрайт-файла
50 LET t=INT(spst/256) : POKE 62464, (spst—256*t): POKE 62465, t:
   REM Указание адреса начала области спрайтов
60 POKE 62466,255 : POKE 62467,220: REM Указание адреса конца
   области спрайтов
70 POKE 56575,0: REM Метка конца области спрайтов
80 LOAD "NAMEcomp": REM Загрузка откомпилированной прог-
   раммы
```

Для работы с этим загрузчиком необходимо только задать конкретные значения:

в строке 20 — адрес нижней границы спрайт-файла (START);
в строке 40 — имя спрайт-файла;
в строке 80 — имя откомпилированной программы.

Во многих случаях загрузчик можно (и нужно) упростить.

Вариант 1. Если в программе не используются спрайты и нет кодовых частей, то загрузчик сократится до 4 строк:

```
10 CLEAR 59799
20 LOAD "RT CODE" CODE
30 POKE 56575,0
40 LOAD "NAMEcomp"
```

Вариант 2. При необходимости подгрузки спрайт-файла, например, с именем "SPRITES" и стартовым адресом 40000, загрузчику можно придать «конкретный» вид:

```
10 CLEAR 39999: REM START—1
20 LOAD "RT CODE" CODE
30 LOAD "SPRITES" CODE 40000
40 POKE 62464,64: POKE 62465,156: REM 40000=64+256×156
50 POKE 62466,255: POKE 62467,220: REM 56575=255+256×220
60 POKE 56575,0
70 LOAD "NAMEcomp"
```

Вариант 3. Может случиться, что для работы откомпилированной программы необходимы кодовые блоки, например, второй знакогенератор или подпрограмма в машинных кодах. Чтобы найти для них место, рассмотрим верхнюю часть памяти ZX Spectrum (табл. 11).

---

* Адрес RAMTOP лучше выбрать на 1 меньше нижней границы спрайт-файла (**START**).

Таблица 11: **Распределение памяти в присутствии откомпилированной программы.**

P_RAMT (23732)	————————————	65535
	Процедуры компилятора "RT CODE"	
	————————————	59800
	Свободная область	
END	————————————	56575
	Область спрайтов	
START	————————————	
RAMTOP=START-1 (23730)	————————————	
	...	
E_LINE (23641)	————————————	
	Переменные Бейсика	
VARS (23627)	————————————	
	Бейсик-программа	
PROG (23635)	————————————	

Как видно из приведенной таблицы, между областью спрайтов и пакетом процедур "RT CODE" имеется около 3 килобайт свободной памяти, начиная с адреса 56576. Проще всего поместить кодовые части именно в эту область. При этом в текст загрузчика (модифицируем 2-й вариант) добавим строку:

```
15 LOAD "YOURCODE" CODE 56576: REM Загрузка кодового файла
```

Несколько сложнее размещать кодовые части, если их длина превышает 3 килобайта. Место под кодовые части можно зарезервировать, вставив в спрайт-файл при его формировании пустой спрайт, то есть спрайт, не несущий никакой информации. Размер и адрес начала пустого спрайта можно определить при его создании в генераторе спрайтов или с помощью функции ?TST. А строка 15 изменится на

```
15 LOAD "YOURCODE" CODE addr
```

– где addr — адрес начала пустого спрайта. При необходимости можно создать несколько пустых спрайтов.

Можно воспользоваться также оператором перемещения спрайт-файла .RLCT. В этом случае в загрузчике нужно дополнительно изменить адрес верхней границы спрайт-файла END.

Выполнение откомпилированной программы

После запуска откомпилированной программы ее невозможно остановить нажатием клавиш **CS/Space** (**Break**). Если же происходит сбой в работе программы, то она прерывается с выдачей одного из стандартных сообщений об ошибке, но, к сожалению, без указания места возникновения ошибки.

Единственное новое сообщение об ошибке, которое генерирует сам пакет рабочих процедур компилятора, — это Program not compiled, что означает: при старте программы она не идентифицирована как скомпилированная.

Корректировать откомпилированную программу, как уже говорилось, можно только одним способом — вносить изменения в текст исходной (некомпилированной) программы и компилировать вновь.

ПРИЛОЖЕНИЯ

1. Операторы и функции Laser Basic

Таблица 12.

Ключевое слово	Действие	Граф. переменные	Адрес п/п	Код	Стр.
	Преобразование окна экрана				
.INVV	Инвертирование окна экрана	.ROW .COL .HGT .LEN	FC38	01	161
.MIRV	Зеркальное отображение окна экрана		FD26	01	161
.MARV	Зеркальное отображение атрибутов окна экрана		FD32	01	161
.SETV	Установка атрибутов в окне экрана		FF9D	01	161
	Скроллинг окна экрана				
.CLSV	Очистка окна экрана	.ROW .COL .HGT .LEN	FF8B	01	161
.WL1V	Циклический скроллинг окна экрана на 1 пиксель влево		F75C	01	161
.WR1V	Циклический скроллинг окна экрана на 1 пиксель вправо		F757	01	161
.SL1V	Скроллинг окна экрана на 1 пиксель влево		F752	01	161
.SR1V	Скроллинг окна экрана на 1 пиксель вправо		F74D	01	161
.WL4V	Циклический скроллинг окна экрана на 4 пикселя влево		F7A5	01	161
.WR4V	Циклический скроллинг окна экрана на 4 пикселя вправо		F785	01	161
.SL4V	Скроллинг окна экрана на 4 пикселя влево		F773	01	161

Ключевое слово	Действие	Граф. переменные	Адрес п/п	Код	Стр.
.SR4V	Скроллинг окна экрана на 4 пикселя вправо	.ROW .COL .HGT .LEN	F761	01	161
.WL8V	Циклический скроллинг окна экрана на 8 пикселей влево		F7CB	01	161
.WR8V	Циклический скроллинг окна экрана на 8 пикселей вправо		F7EA	01	161
.SL8V	Скроллинг окна экрана на 8 пикселей влево		F7BA	01	161
.SR8V	Скроллинг окна экрана на 8 пикселей вправо	.ROW .COL .HGT .LEN	F7D7	01	161
.WCRV	Циклический вертикальный скроллинг окна экрана	.ROW .COL .HGT .LEN .NPX	F65D	0E	162
.SCRV	Вертикальный скроллинг окна экрана		F6CC	0E	162
	Скроллинг атрибутов окна экрана				
.ATLV	Скроллинг атрибутов окна экрана влево	.ROW .COL .HGT .LEN	F7F5	01	162
.ATRV	Скроллинг атрибутов окна экрана вправо		F818	01	162
.ATUV	Скроллинг атрибутов окна экрана вверх		F82A	01	162
.ATDV	Скроллинг атрибутов окна экрана вниз		F874	01	162
	Преобразование спрайтов в памяти				
.INVM	Инвертирование спрайта	.SPN	FC3E	13	167
.MIRM	Зеркальное отображение спрайта		FD2C	13	167
.MARM	Зеркальное отображение атрибутов		FD4F	13	167
.SPNM	Поворот спрайта на 90°	.SP1 .SP2	F9DD	1C	168
.DSPM	Увеличение размеров спрайта		FDB3	1C	168
.SETM	Установка атрибутов спрайта	.SPN	FFBB	13	167
.CLSM	Очистка спрайта		FEEA	13	167
	Скроллинг спрайтов в памяти				
.WL1M	Циклический скроллинг спрайта на 1 пиксель влево	.SPN	F8C7	13	167
.WR1M	Циклический скроллинг спрайта на 1 пиксель вправо		F8CC	13	167

Ключевое слово	Действие	Граф. переменные	Адрес п/п	Код	Стр.
.SL1M	Скроллинг спрайта на 1 пиксель влево	.SPN	F891	13	167
.SR1M	Скроллинг спрайта на 1 пиксель вправо		F8B5	13	167
.WL4M	Циклический скроллинг спрайта на 4 пикселя влево		F900	13	167
.WR4M	Циклический скроллинг спрайта на 4 пикселя вправо		F8F7	13	167
.SL4M	Скроллинг спрайта на 4 пикселя влево		F8EE	13	167
.SR4M	Скроллинг спрайта на 4 пикселя вправо		F8E5	13	167
.WL8M	Циклический скроллинг спрайта на 8 пикселей влево		F8DB	13	167
.WR8M	Циклический скроллинг спрайта на 8 пикселей вправо		F8E0	13	167
.SL8M	Скроллинг спрайта на 8 пикселей влево		F8D1	13	167
.SR8M	Скроллинг спрайта на 8 пикселей вправо		F8D6	13	167
.WCRM	Циклический вертикальный скроллинг спрайта	.SPN .NPX	F94B	17	167
.SCRM	Вертикальный скроллинг спрайта		F91E	17	167
	Скроллинг атрибутов спрайтов в памяти				
.ATLM	Скроллинг атрибутов спрайта влево	.SPN	FC9F	13	167
.ATRM	Скроллинг атрибутов спрайта вправо		FCAA	13	167
.ATUM	Скроллинг атрибутов спрайта вверх		FCAF	13	167
.ATDM	Скроллинг атрибутов спрайта вниз		FCC8	13	167
	Разрешение/запрещение переноса атрибутов				
.ATON	Разрешение переноса атрибутов	—	FA05	00	160
.ATOF	Запрещение переноса атрибутов		FA0D	00	160
	Перемещения: окно экрана — спрайт				
.GTBL	Копирование окна экрана в спрайт	.SPN .ROW .COL	F286	33	166
.GTOR	Наложение окна экрана на спрайт по принципу OR		F28E	33	166
.GTXR	Наложение окна экрана на спрайт по принципу XOR		F296	33	166
.GTND	Наложение окна экрана на спрайт по принципу AND		F29E	33	166

Ключевое слово	Действие	Граф. переменные	Адрес п/п	Код	Стр.
	Перемещения: спрайт — окно экрана				
.PTBL	Вывод спрайта на экран	.SPN .ROW .COL	F2A6	33	160
.PTOR	Наложение спрайта на окно экрана по принципу OR		F2AE	33	164
.PTXR	Наложение спрайта на окно экрана по принципу XOR		F2B6	33	164
.PTND	Наложение спрайта на окно экрана по принципу AND	.SPN .ROW .COL	F2BE	33	164
	Перемещения: окно экрана — окно спрайта				
.GWBL	Копирование окна экрана в окно спрайта	.SPN .ROW .COL .HGT .LEN .SCL .SRW	F2C6	2B	166
.GWOR	Наложение окна экрана на окно спрайта по принципу OR		F2CC	2B	166
.GWXR	Наложение окна экрана на окно спрайта по принципу XOR		F2D2	2B	166
.GWND	Наложение окна экрана на окно спрайта по принципу AND		F2D8	2B	166
	Перемещения: экран — окно спрайта				
.PWBL	Вывод окна спрайта на экран	.SPN .ROW .COL .HGT .LEN .SCL .SRW	F2DE	2B	160
.PWOR	Наложение окна спрайта на экран по принципу OR		F2E4	2B	165
.PWXR	Наложение окна спрайта на экран по принципу XOR		F2EA	2B	165
.PWND	Наложение окна спрайта на экран по принципу AND		F2F0	2B	165
	Перенос атрибутов: экран — спрайт				
.GWAT	Перенос атрибутов окна экрана в окно спрайта	.SPN .ROW .COL .HGT .LEN .SCL .SRW	FB4F	2B	166
.PWAT	Перенос атрибутов окна спрайта в окно экрана		FB5A	2B	160
	Перемещения: спрайт — окно спрайта				
.GMBL	Копирование спрайта в окно другого спрайта	.SP1 .SP2 .SCL .SRW	E985	23	168

Ключевое слово	Действие	Граф. переменные	Адрес п/п	Код	Стр.
.GMOR	Наложение одного спрайта на окно другого по принципу OR	.SP1 .SP2 .SCL .SRW	E991	23	168
.GMXR	Наложение одного спрайта на окно другого по принципу XOR		E997	23	168
.GMND	Наложение одного спрайта на окно другого по принципу AND		E98B	23	168
.PMBL	Копирование окна спрайта в другой спрайт		E99D	23	168
.PMOR	Наложение окна спрайта на другой спрайт по принципу OR		E9A9	23	168
.PMXR	Наложение окна спрайта на другой спрайт по принципу XOR		E9AF	23	168
.PMND	Наложение окна спрайта на другой спрайт по принципу AND		E9A3	23	168
	Перенос атрибутов: спрайт — спрайт				
.GMAT	Перенос атрибутов спрайта в окно другого спрайта	.SP1 .SP2 .SCL .SRW	FC28	23	168
.PMAT	Перенос атрибутов окна спрайта в другой спрайт		FC33	23	168
	Перемещение спрайта по экрану				
.MOVE	Перемещение спрайта	.SP1 .SP2 .ROW .COL .HGT .LEN	—	—	165
	Операции с областью спрайт-файла				
.ISPR	Создание спрайта заданных размеров (понижается нижняя граница спрайт-файла)	.SPN .HGT .LEN	EC43	2B	171
.SPRT	Создание спрайта заданных размеров (повышается верхняя граница спрайт-файла)		EC48	2B	171
.DSPR	Уничтожение спрайта (повышается нижняя граница спрайт-файла)	.SPN	FE80	13	171
.WSPR	Уничтожение спрайта (понижается верхняя граница спрайт-файла)		F52F	13	171
.RLCT	Перемещение спрайт-файла в памяти	.MLN	—	—	172

Ключевое слово	Действие	Граф. переменные	Адрес п/п	Код	Стр.
Вспомогательные графические операторы и функции					
.ADJM	Настройка переменных .HGT и .LEN для спрайта	.SPN .ROW .COL	EBF7	33	172
.ADJV	Настройка переменных .HGT и .LEN для окна экрана	.ROW .COL .HGT .LEN	EC11	01	172
?SCV	Проверка знакоместа на включенные пиксели	.ROW .COL	–	–	172
?SCM	Проверка спрайта на включенные пиксели	.SPN	–	–	172
?TST	Определение существования спрайта		–	–	173
Сервисные операторы					
?KBF	Проверка клавиши на нажатие	.ROW .COL	–	–	174
?PEK	16-битовая версия PEEK	–	–	–	175
.POKE	16-битовая версия POKE	–	–	–	175
.RNUM	Перенумерация строк	–	–	–	175
.REMK	Удаление ремарок	–	–	–	175
.TRON	Включение трассировки	–	–	–	175
.TROF	Выключение трассировки	–	–	–	175

2. Использование графических переменных .SP1 и .SP2

Таблица 13.

Переменная	Оператор			
	.GMBL .GMOR .GMXR .GMND	.PMBL .PMOR .PMXR .PMND	.SPNM	.DSPM
.SP1	откуда	куда	откуда	куда
.SP2	куда	откуда	куда	откуда

3. Листинг программы Spriter*

```
  0 REM © Piter Ltd. Автор П. Карпов
 10 GO SUB 1000
 20 POKE 56575,0: CLS : LET h=1: LET k=1: LET n=1: RANDOMIZE
    USR 58841: GO TO 30
 30 .SET=1:.ROW=0:.COL=0:.HGT=22:.LEN=32
 40 .SET=2:.ROW=0:.COL=0:.SPN=1
 50 .SET=3:.ROW=22:.COL=0:.HGT=2:.LEN=32
 60 INPUT "Screen name? "; LINE a$: LET z=16384: LET x=6912:
    GO TO 2000
 70 LET s=?PEK62464: LET g=56575-s
 80 .SET=3:.SETV:.CLSV: PRINT #1;AT 0,0;".SPN=";n;" .LEN=";k;"
    .HGT=";h'"Start: ";s;" Lenght: ";g
 90 .SET=1
100 PAUSE 0: LET a=CODE INKEY$: GO TO 100+a*(a=98 OR a=99 OR
    a=100 OR a=103 OR a=108 OR a=110 OR a=112 OR a=115 OR
    a=118 OR a>52 AND a<57 OR a=8 AND k>1 OR a=9 AND k<32
    OR a=10 AND h<22 OR a=11 AND h>1)
108 LET k=k-1: GO TO 500
109 LET k=k+1: GO TO 500
110 LET h=h+1: GO TO 500
111 LET h=h-1: GO TO 500
153 .WL8V:.ATLV: GO TO 500
154 .NPX=-8:.WCRV:.ATDV: GO TO 500
155 .NPX=8:.WCRV:.ATUV: GO TO 500
156 .WR8V:.ATRV: GO TO 500
198 STOP
199 GO TO 60
200 .SET=2: LET e=?TST: IF NOT e THEN LET a$="Sprite not
    found":.LEN=k:.HGT=h: GO TO 2050
201 .DSPR: LET a$="Sprite deleted": LET i=4: GO TO 2060
203 .SET=2: LET e=?TST: IF NOT e THEN .LEN=k:.HGT=h:.ISPR:.GTBL:
    LET a$="Sprite gotten": LET i=4: GO TO 2060
204 LET a$="Sprite exist": GO TO 2050
208 INPUT "Starting address? ";z: IF z<30000 OR z>56574 THEN GO TO 70
209 .POKE62464,z: LET x=56575-z: INPUT "Sprite name? "; LINE a$:
    GO TO 2000
210 INPUT ".SPN="; LINE a$: IF a$="" THEN GO TO 70
211 GO TO 300
212 .SET=2: LET e=?TST: IF NOT e THEN LET a$="Sprite not
    found":.LEN=k:.HGT=h: GO TO 2050
213 .PTBL: GO TO 500
215 INPUT "Spritefile name? "; LINE a$: IF a$="" OR NOT g THEN GO
    TO 70
216 IF d THEN SAVE a$CODE s,g: GO TO 70
```

* Внимание! Недопустимо изменять нумерацию строк.

```
217 GO TO 250
218 .SET=2:.LEN=k:.HGT=h:.INVV: BEEP .01,30
220 IF INKEY$<>"" THEN GO TO 220
230 .INVV: GO TO 70
250 LET e=USR 15619: REM : SAVE a$CODE s,g
260 IF NOT e THEN GO TO 70
270 LET a$="File not saved": GO TO 2050
300 FOR i=1 TO LEN a$: IF a$(i)<"0" OR a$(i)>"9" THEN GO TO 70
310 NEXT i: LET m=VAL a$: IF m<=0 OR m>255 THEN GO TO 70
320 LET n=m
500 .SET=2:.SPN=n:.LEN=k:.HGT=h:.INVV: BEEP .01,30:.INVV: GO TO 70
1000 POKE 23658,0: CLS : PRINT AT 3,6;"Tape or Disk? (T/D)"
1010 PAUSE 0: IF INKEY$="t" THEN CLS : LET d=1: GO TO 1040
1020 IF INKEY$<>"d" THEN GO TO 1010
1030 LET d=0
1040 CLS : BEEP .005,30: RETURN
2000 IF a$="" THEN GO TO 70
2010 IF d THEN LOAD a$CODE z,x: GO TO 70
2020 LET e=USR 15619: REM : LOAD a$CODE z,x
2030 IF NOT e THEN GO TO 70
2040 LET a$="File not found"
2050 LET i=2
2060 PRINT #1;AT 1,0; INK i; FLASH 1;a$: BEEP 1,3: PAUSE 100: GO TO 70
3000 INK 7: PAPER 0: BORDER 0: CLEAR 29999: GO SUB 1000
3010 IF d THEN LOAD "LASOBJ"CODE : LOAD "LASLOBJ"CODE :
     RANDOMIZE USR 62464: LOAD "GRAPH"CODE : GO TO 20
3020 LET e=USR 15619: REM : LOAD "LASOBJ"CODE
3030 IF NOT e THEN LET e=USR 15619: REM : LOAD "LASLOBJ"CODE
3040 IF NOT e THEN RANDOMIZE USR 62464: LET e=USR 15619: REM :
     LOAD "GRAPH"CODE
3050 IF NOT e THEN GO TO 20
3060 PRINT #1;AT 1,0; INK 2; FLASH 1;"Loading error": BEEP 1,3:
     PAUSE 100: STOP
```

Программа сохраняется оператором

```
SAVE "SPRITER" LINE 3000
```

На магнитном носителе вслед за файлом "SPRITER" должны быть записаны следующие кодовые файлы: "LASOBJ", "LASLOBJ" и "GRAPH" (см. Приложение 5).

4. Вызов подпрограмм Laser Basic из машинных кодов*

Интерпретатор Laser Basic можно использовать в качестве пакета рабочих процедур при написании программ в кодах. Адреса процедур Laser Basic хранятся в памяти в виде таблицы, начинающейся с адреса #EC4D**. В ней на каждый оператор отводится 8 байт. Первые 4 байта — имя оператора, следующие 2 байта — собственно адрес, 7-й байт — код типа команды, последний байт — порядковый номер оператора в таблице.

Графические переменные хранятся в виде наборов (.SET). На каждый набор отводится 12 байт, начиная с адреса #E7CB (59339). Адрес конкретного набора вычисляет подпрограмма #E96E. Перед обращением к ней в регистр A заносится номер набора (0...15)***, а в HL подпрограмма возвращает требуемый адрес (его лучше представлять через регистровую пару IX):

```
LD      A,<номер набора>
CALL    #E96E
PUSH    HL
POP     IX
```

Переменные в каждом наборе располагаются в следующей последовательности:

IX+0	младший байт .MLN	IX+6	.SRW
IX+1	старший байт .MLN	IX+7	.SCL
IX+2	.ROW	IX+8	.NPX
IX+3	.COL	IX+9	.SPN
IX+4	.LEN	IX+10	.SP1
IX+5	.HGT	IX+11	.SP2

В табл. 12 (Приложение 1) и табл. 14 приведена вся необходимая информация для непосредственного обращения к подпрограммам Laser Basic. В первой таблице находим интересующий оператор и определяем адрес процедуры и код типа команды, по которому из второй таблицы узнаем способ обращения к данной процедуре. Например, нам нужно выполнить оператор .SETV. В табл. 12 находим адрес — #FF9D, код операции — #01. Из табл. 14 узнаем, что перед обращением к подпрограмме регистры B, C, H и L должны быть загружены, соответственно, значениями .ROW, .COL, .HGT и .LEN. Кроме того, поскольку оператор .SETV работает с атрибутами, предварительно нужно занести (если требуется, конечно) в системную переменную ATTR_P (23693) байт атрибутов. После этого можно вызвать подпрограмму, которая установит атрибуты в заданном окне. Выглядит это, например, так:

```
10    LD    A, %01101001      ; задаем INK 5, PAPER 1, BRIGHT 1
20    LD    (23693), A
```

* Предлагаем этот раздел для читателей, знакомых с программированием на ассемблере.

** В этом приложении адреса ячеек памяти приведены в шестнадцатеричном виде.

*** Если используется текущий набор, то его номер можно взять из ячейки #E902.

```
30 ;
40 ;загружаем регистры значениями  .ROW=1, .COL=7, .HGT=4, .LEN=5
50        LD     B, 1
60        LD     C, 7
70        LD     H, 4
80        LD     L, 5
90        CALL   #FF9D           ; выполняем оператор .SETV
```

Таблица 14. **Загрузка регистров процессора при обращении к процедурам Laser Basic.**

Код оператора	Соответствие: регистры — переменные
00	–
01	B=.ROW, C=.COL, H=.HGT, L=.LEN
0E	A=.NPX, B=.ROW, C=.COL, H=.HGT, L=.LEN
13	A=.SPN
17	A=.SPN, B=.NPX
1C	B=.SP1, C=.SP2
23	B=.SP2, C=.SP1, D=.SRW, E=.SCL
33	A=.SPN, B=.ROW, C=.COL

5. Спецификация файлов пакета Laser Basic*

Таблица 15.

Название	Имя файла	Тип файла	Длина	Адрес загрузки
Диспетчер	LASER	BASIC	7540	–
Интерпретатор	LASOBJ	CODE	3642	58820
	LASLOBJ	CODE	1024	62464
	GRAPH	CODE	3072	62464
Спрайт-файлы для интерпретатора	SPRITE2A	CODE	5062	51513
	SPRITE2B	CODE	6271	50304

* Приводится спецификация фирменного пакета. В других версиях могут отличаться имена и длины бейсик-файлов.

Название	Имя файла	Тип файла	Длина	Адрес загрузки
Генератор спрайтов	SPTGEN	BASIC	20751	–
	CODE	CODE	3000	49898
Спрайт-файлы для генератора спрайтов	SPRITE1A	N. ARRAY	1288	–
	SPRITE1A	CODE	5062	60218
	SPRITE1A	CODE	6	23307
	SPRITE1B	N. ARRAY	1288	–
	SPRITE1B	CODE	6272	59009
	SPRITE1B	CODE	6	23307
Демонстрационные программы	DEMO	BASIC	358	–
	SCREEN	CODE	6912	16384
	SPRITES	CODE	6852	49723
	BASIC	BASIC	23790	–
	GAME	BASIC	352	–
	SCREEN	CODE	6912	16384
	SPRITES	CODE	2795	53780
	INVADER	BASIC	15330	–
Компилятор	COMPCODE	CODE	3057	59800
Загрузчик откомпилированных программ	LOADER	BASIC	223	–
	RTCODE	CODE	5736	59800

MEGABASIC

YS MegaBasic — один из лучших диалектов Бейсика для ZX Spectrum — был распространен в 1985 году под эгидой журнала «Your Sinclair» (от названия журнала и возникли первые буквы в имени программы).

Имеют хождение три версии MegaBasic, датированные все 1985 годом: 1.1, 3.0 и 4.0. Различия между версиями 1.1 и 3.0 скорее косметические, причем не в пользу версии 3.0. Дело в том, что MegaBasic 3.0 при попытке ввода некорректно сформулированных строк вместо выдачи предусмотренного на этот случай сообщения **Syntax error** имеет обыкновение время от времени «зависать». Поэтому пользоваться этой версией без особых на то оснований не рекомендуется. Зато в версии 4.0 проблема проверки вводимого текста решена кардинально: синтаксис редактором почти не проверяется, и комментирует программистские изыски MegaBasic 4.0 уже в ходе выполнения программы. Это объясняется тем, что редактор MegaBasic 4.0 разрешает сокращенную форму ввода не только для операторов стандартного Бейсика, как в предыдущих версиях, но и для многих операторов самого MegaBasic. С этим связано и единственное ограничение на совместимость различных версий: программы, составленные для MegaBasic 4.0 с сокращенной записью операторов MegaBasic, нельзя использовать в более ранних версиях.

Дальнейшее изложение, если не оговорено иное, справедливо для любой версии MegaBasic. Однако для большей наглядности мы будем работать с версией 4.R — русифицированным MegaBasic 4.0*.

Поскольку MegaBasic вышел на рынок в 1985 году, а первый 128-килобайтный Spectrum — годом позже, то они, естественно, оказались несовместимы, о чем остается лишь сожалеть, так как

---

* Автор Mike Leaman. Файлы версии 4.0: MegaBasic (тип BASIC, длина 655 байт), MegaCode (тип CODE, длина 20373 байт, адрес загрузки 44996). О том, как и где приобрести версию 4.R, смотрите в рекламном разделе книги.

из-за большого собственного объема MegaBasic (свыше 20 килобайт) часто ощущается нехватка памяти при создании программ.

Второй недостаток MegaBasic — несовместимость с дисковой системой TR-DOS. Причина в том, что MegaBasic, так же, как и TR-DOS, использует 2-й режим прерываний процессора*. Однако существуют способы преодоления этого недостатка. Один из них, наиболее простой, приведен в Приложении 3.

ЧТО МОЖЕТ MEGABASIC?

Прежде чем начать работу с любой системной программой, нелишне выяснить ее возможности и сопоставить их со своей задачей. Поэтому кратко расскажем об особенностях MegaBasic, отталкиваясь от стандартного Spectrum-Бейсика.

Прежде всего, MegaBasic может абсолютно все, на что способен и Spectrum-Бейсик, если это «все» не требует больше 20 килобайт свободной памяти. И, конечно, любая программа на стандартном Бейсике (с учетом указанного ограничения по объему), будучи загружена в MegaBasic, станет прекрасно работать**.

Графические возможности MegaBasic чрезвычайно богаты. Во-первых, это наличие многооконного интерфейса, позволяющего достаточно гибко манипулировать выводом информации на экран. Во-вторых, три встроенных символьных набора с изменяемыми в широких пределах размерами символов. Наконец, оригинальный механизм управления спрайтами и кадрами, что существенно для написания игровых программ. Все это, несомненно, воодушевит тех программистов, которые любят «делать красиво».

Звуковые способности MegaBasic настолько же превосходят возможности оператора BEEP Spectrum-Бейсика, насколько и уступают потенциалу музыкального процессора компьютера ZX Spectrum 128.

Что касается достижений MegaBasic в технике программирования, то в первую очередь обращает на себя внимание довольно удобный редактор с посимвольным вводом операторов и возможностью копирования любого отрезка выведенного на экран текста в редактируемую строку. Разумеется, редактор дополнен большим числом новых команд, предназначенных как для управления многочисленными дополнительными функциями MegaBasic, так и для создания дружественной обстановки для программиста.

---

* О прерываниях читайте в [1].

** Неработоспособными окажутся только те программы, внутри которых (скажем, под оператором **REM**) расположены подпрограммы в машинных кодах с абсолютной адресацией, так как начало бейсик-программ MegaBasic устанавливает на 11 байт выше, чем это делает стандартный интерпретатор. Кроме того, в бейсик-программе не должно быть обращений к подпрограммам в кодах, расположенным выше адреса 45000 (исключая область UDG).

В большом количестве MegaBasic содержит и средства структурного программирования: это и традиционный для расширений Spectrum-Бейсика инструмент процедур, и циклы типа REPEAT...UNTIL, и операторы, обеспечивающие параллельное выполнение двух программ, и пр.

MegaBasic заботится и о тех, кто любит «копнуть поглубже»: кроме дополнительных команд работы с машинными кодами, он содержит встроенный монитор-отладчик.

РЕДАКТОР

AUTO, DELETE, EDIT, ESCAPE, RESET

Как ни крути, но приступать к программированию на MegaBasic можно лишь после ознакомления с работой его редактора, тем более, что он кардинально отличается от стандартного спектрумовского.

Все ключевые слова в MegaBasic набираются посимвольно, как обычный текст. При этом можно использовать как прописные, так и строчные буквы: все, что не заключено в кавычки или не стоит под REM, редактор сам преобразует в прописные. Предусмотрена возможность ввода ключевых слов и в сокращенной форме, завершая сокращенное слово точкой. В версиях MegaBasic 1.1 и 3.0 сокращения допускаются только для стандартных операторов Spectrum-Бейсика и должны формулироваться однозначно. Версия 4.0 допускает сокращенную запись и новых операторов, а также различные варианты сокращений. Например, для оператора GO TO в версиях 1.1 и 3.0 принято только сокращение G., а в версии 4.0 возможны варианты G., GO ., GO.. Правильно сокращенные операторы Spectrum-Бейсика редактор преобразует в соответствующий код синклеровской таблицы символов и вывод листинга программы осуществляет уже без сокращений.

В собственных операторах MegaBasic связкой между ключевым словом и параметрами, если таковые предусмотрены, служит символ подчеркивания. Он должен присутствовать и в сокращенной записи ключевых слов MegaBasic (для версии 4.0). Например, вместо EDIT_10 можно писать ED._10.

Если есть намерение написать достаточно длинную программу, то полезна будет команда

AUTO_n,k

Она осуществляет автоматическую нумерацию строк. После выполнения AUTO_n,k в режиме ввода редактор будет подсказывать номера очередных строк, начиная с номера n и с интервалом k. Действие AUTO прекращается нажатием комбинации клавиш CS/SS+SS/L. Надо следить только за тем, чтобы номер строки не превысил 9999, иначе возможны трудности с выходом из режима AUTO.

Команда

DELETE_n1,n2

удаляет блок программы. Параметры n1 и n2 — номера первой и последней удаляемых строк.

Вызвать на редактирование ранее введенную строку можно как обычно: «опустить» ее из листинга в строку редактора с помощью клавиш **CS/1**. Но иногда для этого удобнее воспользоваться специальной командой

EDIT_n

– указав номер нужной строки n.

Обратите внимание, что в MegaBasic вертикальное перемещение курсора при переходе к соседней строке листинга программы приводится в действие одновременным нажатием клавиш **SS** и **U** (вверх) или **SS** и **Y** (вниз). Обычный же способ управления движением курсора вверх-вниз используется для вертикальных перемещений его в пределах вызванной на редактирование строки, что очень полезно при редактировании длинных (более 32 символов) строк.

В MegaBasic введен ряд дополнительных управляющих клавиш, работающих при редактировании строки программы:

- **SS/Q** – устанавливает курсор в начало строки;
- **SS/E** – устанавливает курсор в конец строки;
- **CS/3** – удаляет всю редактируемую строку;
- **CS/4** – удаляет символ правее курсора;
- **SS/W** – удаляет текст от курсора до конца строки.

Ввод отредактированной строки происходит, естественно, при нажатии клавиши **Enter**. После чего в версиях 1.1 и 3.0 MegaBasic либо принимает строку, либо с сообщением Syntax error (синтаксическая ошибка) или Bad line (неправильная строка) отказывает в этом. Версия 4.0, как правило, принимает результат трудов программиста безоговорочно, а если он допустил ошибку, то соответствующее сообщение выдается уже в процессе выполнения программы.

Следует также сказать о весьма удобном способе формирования редактируемой строки с помощью *копирующего курсора*. Выведите на экран какой-либо текст, например, листинг программы. Нажмите теперь **SS** вместе с одной из клавиш: **A**, **S**, **D** или **F**. В углу экрана появится зеленый квадратик — это и есть копирующий курсор. Упомянутыми клавишами, постоянно удерживая **SS**, его можно подвести к нужному месту экрана. Если теперь, не отпуская клавишу **SS**, нажать и держать клавишу **I**, то копирующий курсор побежит вдоль выбранной строки и перенесет текст символ за символом в редактируемую строку. Эту операцию можно комбинировать с вводом символов с клавиатуры обычным порядком.

Нужно следить за тем, чтобы шрифт, используемый в данный момент редактором, был из того же символьного набора, что и копируемый текст, а размер шрифта был стандартным (8×8 пикселей), иначе копирование невозможно.

Предусмотрены также дополнительные сочетания клавиш для перемещения копирующего курсора: комбинация **SS/CS+SS/M** переводит копирующий курсор в верхний левый угол текущего окна (об окнах позже), а **SS/CS+SS/N** переносит копирующий курсор в следующее окно (этот эффект не очевиден, если окна совпадают).

Листинг программы можно вывести на экран как обычной командой LIST, так и нажатием комбинации клавиш **SS/CS+SS/K**. Нужно иметь в виду, что при выводе листинга, занимающего больше одного экрана, он прокручивается до конца: редактор MegaBasic не выдает сообщения scroll?. Однако вывод можно приостановить, нажав и удерживая клавишу **M**. После ее освобождения прокрутка возобновится. Если же нужно прекратить вывод информации, можно воспользоваться командой

ESCAPE

Вызывается она одновременным нажатием клавиши **Space** и **E**. Кроме прерывания вывода листинга, команда ESCAPE, как и **Break**, прекращает выполнение бейсик-программ. Более сильный вариант останова любой программы — команда

RESET

Она инициируется одновременным нажатием клавиш **Space** и **R** (не путать с кнопкой RESET на компьютере, это будет слишком уж сильно!). Выполнение RESET, сохраняя в памяти программу, переводит MegaBasic в исходное состояние. Точнее, в условно исходное, поскольку, к примеру, ни команда RESET, ни даже NEW не восстанавливают исходные параметры окон и некоторые другие значения, задаваемые системой при запуске MegaBasic. В отдельных случаях это доставляет неудобства, например, при загрузке другой программы вместо отработавшей.

ШРИФТЫ

FONT, MODE, VDU, STIPPLE, SPRINT, DOWN

Начнем работу с MegaBasic с самого простого — изучим его шрифтовые возможности. Как уже упоминалось, для подготовки примеров мы будем пользоваться русифицированной версией MegaBasic 4.R. Это означает, что в оригинальной версии 4.0 один из двух латинских символьных наборов (FONT_1) заменен набором символов русского алфавита. В нем очертания знаков составляют стилистическое единство с сохраненным латинским шрифтом (FONT_2). Кроме того, в распоряжении программиста остается стандартный символьный набор, хранящийся в ПЗУ ZX Spectrum (FONT_0).

Итак, мы упомянули оператор, который переключает символьные наборы. Его формат

FONT_i

– где i — номер набора (0, 1 или 2).

Текстовые возможности MegaBasic не ограничиваются выбором одного из трех символьных наборов. Он позволяет менять размер

символов текущего набора: 4×8, 8×8, 8×16 и 16×16 точек (рис. 18). Устанавливаются размеры шрифта оператором

MODE_m,[w]

Параметр m может принимать значения от 1 до 4, соответствующие приведенному ряду размеров шрифтов. Оператор MODE используется и с двумя параметрами: MODE_w,m. В этом формате w обозначает номер окна, для которого устанавливается размер шрифта m.

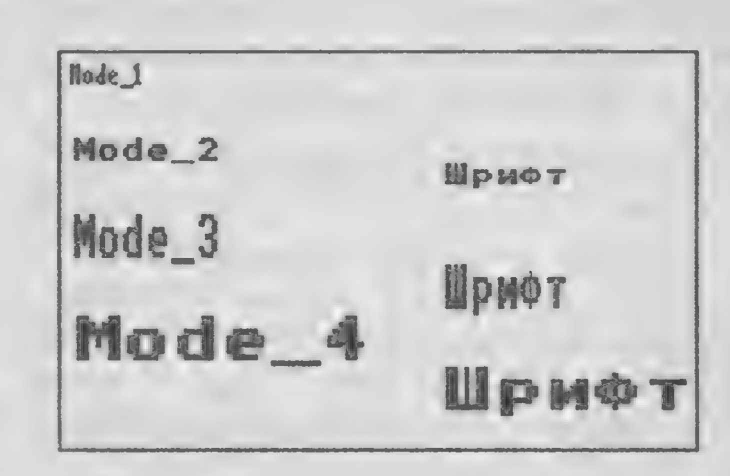


Рис. 18. Шрифты MegaBasic.

Комбинируя названные выше операторы обработки шрифтов, напишем небольшую демонстрационную программу:

```
 10 PAPER 1: INK 6: BRIGHT 1: CLS
 15 FONT_1: LET A$="Шрифты": GO SUB 100
 20 FONT_2: LET A$="Fonts": GO SUB 100
 30 FONT_0: LET A$="Fonts": GO SUB 100
 40 MODE_1: GO TO 15
100 FOR N=1 TO 4:MODE_N: PRINT A$: PRINT : PAUSE 40: NEXT N
110 RETURN
```

Обратите внимание, что для шрифта FONT_1 (кириллица) не выводятся уменьшенные русские буквы в режиме MODE_1. Это объясняется тем, что в этом режиме MegaBasic не генерирует символы, а пользуется содержащимся в нем специальным (четвертым) символьным набором (аналогичным применяемому в одном из популярных текстовых редакторов Tasword).

MegaBasic предусматривает и альтернативный способ управления размерами символов. Оператор стандартного Бейсика PRINT CHR$ с параметрами 1...4 (не используемые Spectrum-Бейсиком коды символов) производит то же действие, что и оператор MODE. Еще один оператор MegaBasic, позволяющий достичь того же эффекта

VDU_m

– представляется уж совершенным излишеством.

В сочетании с режимом FONT_4 может использоваться оператор

STIPPLE_d

– заштриховывающий символы (рис. 19). Плотность штриховки задается параметром d, рабочие значения которого лежат в пределах от 0 («ну-

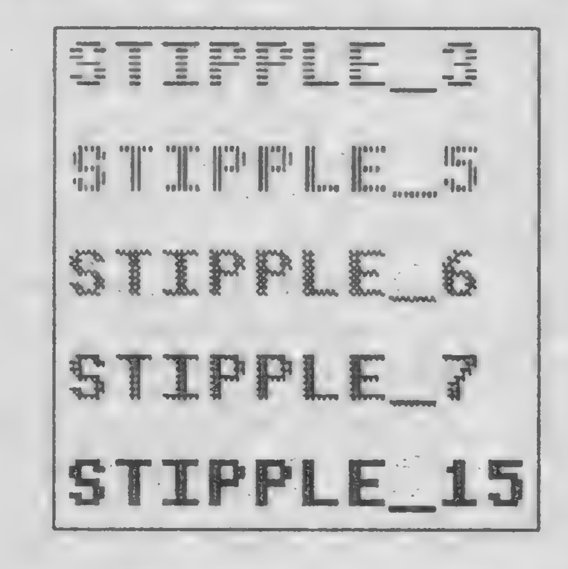


Рис. 19. Варианты штриховки.

левая» штриховка, символ не отличим от фона) до 15 (сплошное закрашивание).

В арсенале MegaBasic есть еще два оператора, нестандартно выводящие тексты на экран. Это SPRINT и DOWN.

SPRINT_x,y,u,v,a$



выводит символьную строку с увеличением в указанное число раз без привязки к знакоместам, то есть начиная с произвольной точки экрана. Параметры x и y задают координаты в пикселях начальной (левой верхней) позиции вывода текста (за начало координат принимается в данном случае левый верхний угол экрана); u и v — масштаб увеличения символа по горизонтальной и вертикальной координатам (дробные значения масштаба округляются до целых); a$ — строку символов, предназначенную для вывода на экран.

DOWN_n,c,a$

тоже распечатывает на экране символьную строку, правда, без масштабирования и с привязкой не к точке, а к знакоместу, но зато вертикально — сверху вниз. Параметры n и c определяют координаты начальной позиции вывода (строка, столбец), а a$ — строковую переменную, предназначенную для вывода.



Теперь существенное замечание. В операторах MegaBasic, имеющих отношение к выводу на экран текстовой информации, размер знакомест (если не оговорено иное) считается равным не 8×8 пикселей, как в стандартном Бейсике, а 4×8*. Соответственно, строки отсчитываются от верха экрана по-прежнему с 0 по 23, а столбцы от левого края экрана, но не с 0 по 31, а с 0 по 63, то есть по половинке стандартного знакоместа.

Следующая программа продемонстрирует возможности SPRINT и DOWN:

```
 10 PAPER 1: INK 6: BRIGHT 1: CLS :FONT_1: LET A$="Шрифт"
100 FOR U=1 TO 6: LET X=(128-20*U): FOR V=1 TO 20 STEP 2
110 SPRINT_X,5,U,V,A$
115 INK 2:MODE_4:DOWN_5,30,A$: INK 6
120 PAUSE 40: CLS : NEXT V: NEXT U
130 GO TO 100
```

* Это связано с возможностью MegaBasic выводить символы размером 4 × 8 точек.

ТЕКСТОВЫЕ ОКНА

WINDOW, CURRENT, FX, CLW

Теперь самое время перейти к описанию возможностей MegaBasic по управлению текстовыми окнами. Сразу после загрузки MegaBasic все окна, кроме окна встроенного в MegaBasic монитора-отладчика (о нем подробный разговор будет позже), совпадают с основным экраном стандартного интерпретатора. MegaBasic позволяет одновременно оперировать десятью окнами.

WINDOW_n,c,h,g

Оператор задает размеры окон. Параметры n (0...23) и c (0...63) устанавливают позицию левого верхнего угла окна, соответственно, по вертикали и горизонтали, а h (1...24) и g (1...64) — его размер (высоту и длину). Координаты задаются в знакоместах.

Оператор WINDOW устанавливает размеры только *текущего* (активизированного в данный момент) окна. Окно номер w делает текущим оператор

CURRENT_w

Возможности оконного интерфейса MegaBasic продемонстрируем на примере программы, которая выдает свой листинг в каждом из активизируемых по очереди окон:

```
4 PAPER 1: CLS
5 FOR N=2 TO 7: PAPER N-2: INK 9-N
10 CURRENT_N:WINDOW_N,4*N,10,34:FX_1,N
20 CLW_N,1: REM Оператор очистки окна (описан чуть ниже)
25 MODE_N,2
30 LIST: LIST: LIST: LIST
40 PAUSE 80: NEXT N
50 GO TO 5
```

Управление окнами (правда, не всеми) можно осуществлять и с помощью оператора PRINT CHR$ с использованием кодов от 24 до 31, которые стандартный Бейсик оставил вакантными. При этом PRINT CHR$ 24 устанавливает текущим окно номер 0, PRINT CHR$ 25 — окно 1 и далее до PRINT CHR$ 31, который включает окно 7. Окна 8 и 9 этим способом не управляются.

Того же самого эффекта можно добиться, используя оператор VDU_24...VDU_31.

Потенциальные возможности оператора

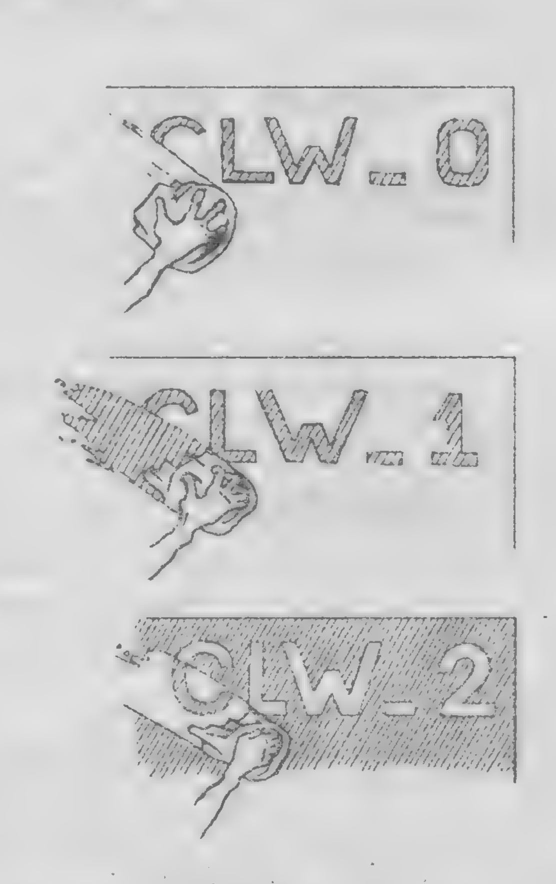
FX_i,w

еще ждут своего исследователя. В фирменном описании указывается, что он служит для распределения потоков информации между окнами. Параметр i определяет вид выводимой информации: 0 — командная строка и сообщения об ошибках; 1 — листинг программы; 2 — информация, генерируемая в процессе выполнения программы; 3 — встроенный монитор-отладчик MegaBasic. Параметр w задает номер окна, в которое будет направляться информация из указанного потока (значения от 0 до 9).

Однако параметр i оператора FX может принимать не только значения от 0 до 3, но и 4, 5, 6. При этом параметр w может иметь значения вплоть до 66063! Это побочные, недокументированные режимы, о которых никакие руководства не упоминают. Потоки FX_4 и FX_5 могут использоваться, вероятно, для управления периферийными устройствами. Что касается FX_6,w, то в сочетании с рабочими значениями параметра w от 0 до 255 его действие аналогично POKE 23609,w, то есть он изменяет длительность звука, подтверждающего нажатие клавиши на клавиатуре.

CLW_[w,]m

осуществляет очистку окна. Если параметр один (m), то действие оператора относится к текущему окну и значение этого параметра определяет способ, которым будет реализована очистка окна:



- 0 – заполнение окна цветом фона (PAPER), аналогично действию оператора CLS Spectrum-Бейсика, но в пределах окна;
- 1 – производит то же действие, что и CLW_0, но с заполнением цветом тона (INK);
- 2 – инвертирует изображение в окне (меняет цвет фона на цвет тона и наоборот);
- 3 – заменяет все атрибуты текущего окна на постоянные.

В случае двух параметров в операторе CLW первый (w) устанавливает номер окна, к которому относится действие оператора. Например, оператор CLW_w,3 заменяет атрибуты окна с номером w на атрибуты, установленные для текущего окна. Необходимо внимательно следить за тем, чтобы значение параметра w не выходило за разрешенные пределы (0...9). Иначе в лучшем случае программа прервется сообщением Y too large, в худшем, но наиболее вероятном — зависнет. Зато второй параметр (он бывает и единственным) можно увеличивать хоть до миллиона, поскольку любое значение свыше трех приравнивается трем.

В заключение рассказа о текстовых окнах приведем эффектную программу с использованием операторов WINDOW и CLW, имитирующую постепенное раскрывание окна:

```
 5 PAPER 0: INK 5: BRIGHT 1: FONT_1:MODE_4
10 CLS: FOR N=1 TO 10
15 LET S=10–N: LET L=33–3*N: LET H=2*N+1: LET W=6*N
20 WINDOW_S,L,H,W:CLW_1
30 NEXT N
40 STIPPLE_(RND*60): PRINT AT 9,15; INVERSE 1; PAPER 1;"O K H O"
50 PAUSE 80: GO TO 10
```

АТРИБУТЫ

INVERT, FADE, SWAP, CHANGE

С одним из операторов, работающих с атрибутами экрана, мы уже встретились в предыдущем разделе. Это оператор CLW, который позволяет очищать окно с заменой атрибутов. Похожие действия выполняют и некоторые другие операторы, предусматривающие, наряду с выполнением своей основной функции, модификацию атрибутов: SCROLL, PAN, GET, PUT. Эти операторы будут рассмотрены в тех разделах описания, где речь пойдет об их основном назначении. Сейчас же остановимся на операторах MegaBasic, занимающихся исключительно обработкой атрибутов: INVERT, FADE, CHANGE и SWAP. Сразу отметим, что действие всех перечисленных операторов распространяется на весь экран, то есть они игнорируют деление экрана на окна.

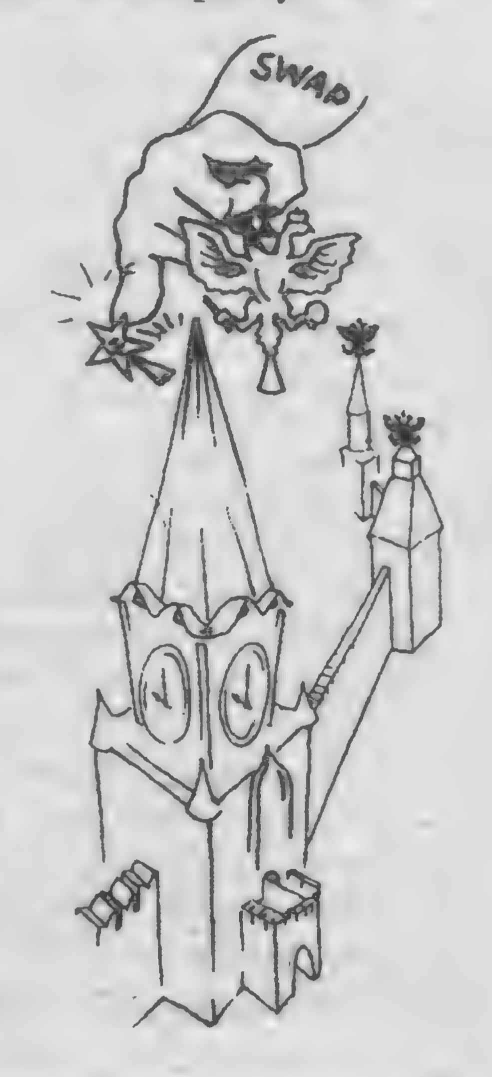
INVERT

инвертирует изображение на экране, заменяя цвет фона на цвет тона и наоборот. Используется без параметров.

Применение оператора

FADE_a

несколько сложнее, но зато действие его гораздо эффектнее. Этот оператор имеет один параметр, который задает значение байта атрибутов. Оператор просматривает значения всех атрибутов экрана* и уменьшает на единицу те из них, которые не совпадают с **a**. Операция выполняется циклически до тех пор, пока все значения атрибутов не сравняются с **a**. Эта процедура вызывает появление на экране довольно интересных цветовых эффектов.



Рабочее значение параметра **a** может находиться в пределах от 0 до 255, но оператор FADE «проглатывает» и любые другие числа, реагируя при этом только на остаток от их деления на 256.

Операторы CHANGE_a1,a2 и SWAP_a2,a1 также осуществляют замену атрибутов, но используют по два параметра, каждый из которых представляет собой значение атрибута.

Оператор

SWAP_a1,a2

служит для замены атрибута **a2** атрибутом **a1**: находит на экране знакоместа с атрибутами, равными значению параметра **a2**, и

---

* Система построения числовых значений атрибутов в MegaBasic не отличается от принятой в Spectrum-Бейсике (см. стр. 55).

устанавливает в них новые атрибуты, соответствующие значению параметра a1. Скажем, SWAP_BIN 00000001,BIN 10000001* включит FLASH в тех знакоместах экрана, в которых изображение состоит из синих точек на черном фоне.

Оператор

CHANGE_a1,a2

производит поиск и замену сразу нескольких атрибутов. Работает CHANGE следующим, довольно нетривиальным, образом. Он изменяет по всему экрану те биты атрибутов, которые не совпадают в параметрах a1 и a2. Так, если воспользоваться теми же значениями параметров, что и в приведенном выше примере с оператором SWAP, то есть выполнить CHANGE_BIN 00000001,BIN 10000001, то FLASH включится по всему экрану. Зато если потребуется, к примеру, перекрасить желтым цветом все, что изображено синим, красным или фиолетовым, независимо от фона, то эту задачу можно решить, введя CHANGE_BIN 00000011,BIN 00000110. Из операторов SWAP пришлось бы в этом случае составлять длинную цепочку, учитывающую все варианты фона.

КАДРЫ

GET, PUT, SPUT

MegaBasic позволяет копировать часть экранного изображения (которую мы будем называть *кадром*) в свободную область памяти (далее именуемую буфером) и возвращать кадр в прежнем или измененном виде на старое или новое место экрана.

Кадр копируется в буфер оператором

GET_m,z,n,c,h,g

– а возвращается на экран оператором

PUT_m,z,n,c,h,g

Форматы операторов GET и PUT, в принципе, схожи.

Параметр m (0...6) задает принцип, по которому кадр совмещается с изображением в буфере (GET) или на экране (PUT). При m=0 копируемое изображение заменяет собой предыдущее. При m=1 происходит логическое сложение (OR) старого и нового содержимого; при m=2 или 3 сложение осуществляется по принципу «исключающего ИЛИ» (XOR). Операции, производимые при m=4, 5 и 6, аналогичны операциям при m=0, 1 или 2, но с той лишь разницей, что при копировании знакоместам в кадре присваиваются текущие постоянные атрибуты.

Параметр m в операторе GET обычно устанавливают равным нулю (иные значения в фирменном руководстве даже не упоминаются). Однако ненулевое значение m может пригодиться, если

* Рекомендуем задавать значения атрибутов в двоичной форме – это более наглядно.

нужно совместить в памяти изображения двух или более кадров. Нужно соблюдать только одно условие — кадры должны быть одинакового размера.

Второй параметр (z) в операторах GET и PUT устанавливает начальный адрес буфера, с которым взаимодействует выделенный кадр. Нужно следить, чтобы буферы не перекрывались и, естественно, не «наезжали» на коды MegaBasic, то есть не оказывались выше адреса 45000.

Остальные четыре параметра задают координаты и размеры кадра в «нормальных» знакоместах — 8×8 пикселей; начало координат — левый верхний угол экрана. Параметры n и c указывают, соответственно, вертикальную и горизонтальную координаты верхнего левого угла кадра. Параметры h и g определяют высоту и ширину кадра. К их значениям в операторе PUT предъявляется жесткое требование — они должны в точности совпадать со значениями параметров h и g, использованными предшествующим оператором GET, иначе возникнут искажения изображения.

Несколько слов о рабочих и допустимых значениях параметров. Рабочими значениями параметра m являются целые числа от 0 до 6. Но и любые другие значения не приводят к ошибке, так как MegaBasic использует только 4 младших бита двоичного представления целой части параметра.

Следующий параметр — адрес буфера z — должен указывать на свободную область памяти. Его рабочие значения обсуждались чуть раньше. Произвольные значения также принимаются операторами GET и PUT и пересчитываются к приемлемым для ZX Spectrum значениям адресов. Однако трудно предположить, где окажется буфер, если указать, к примеру, восьмизначный адрес.

Рабочие значения остальных четырех параметров операторов GET и PUT должны быть такими, чтобы весь кадр попадал в пределы экрана. Допустимые значения вертикальной координаты совпадают с рабочими, а горизонтальной позволяют однократный переход правой части кадра, не поместившегося целиком на экране, на левую сторону экрана. Нулевых значений высоты и ширины кадра операторы GET и PUT не любят (вплоть до зависания программы).

Прежде чем приступить к использованию оператора GET, нужно зарезервировать с помощью CLEAR участок памяти под буфер. Сделать это можно и «с запасом», но в принципе минимальный необходимый объем памяти для буфера (в байтах) легко рассчитывается. Он равен 9-кратному произведению ширины на высоту кадра, выраженных в «старых» знакоместах (8×8 точек).

Пример работы с операторами GET и PUT, несмотря на свое шутливое содержание, на самом деле представляет собой весьма практичную подпрограмму вызова оверлейного меню:

```
10 CLEAR 29999:FONT_1:CURRENT_4: PAPER 0: INK 0:CLW_4,0: LET
   L=4: LET S=4: LET H=12: LET G=24:WINDOW_L,S*2,H,G*2
20 MODE_2: PRINT PAPER 0; INK 0;AT 1,16;"М Е Н Ю" ' '
   "На закуску — Селедочка" ' "На первое — Щи с грибами" '
   "На второе — Бифштекс по-деревенски" '
   "На третье — Бланманже" ' ' "Напитки в ассортименте"
30 PAPER 7:GET_4,30000,L,S,H,G
```

```
40 CURRENT_2:MODE_4:STIPPLE_10: FOR N=1 TO 30: PRINT INK
   (1+RND*6);"Уже несут меню!": NEXT N
50 LET L1=1+RND*4: LET S1=1+RND*4:GET_0,35200,L1,S1,H,G
60 PUT_0,30000,L1,S1,H,G: PAUSE 500
70 PUT_0,35200,L1,S1,H,G
80 PAUSE 120: GO TO 40
```



В MegaBasic существует еще один оператор (SPUT), который как и PUT, переносит кадр из буфера на экран. Но делает он это с изменением масштаба. Оператор SPUT тянет за собой рекордное количество параметров — целых семь:

SPUT_z,x,y,u,v,g,h

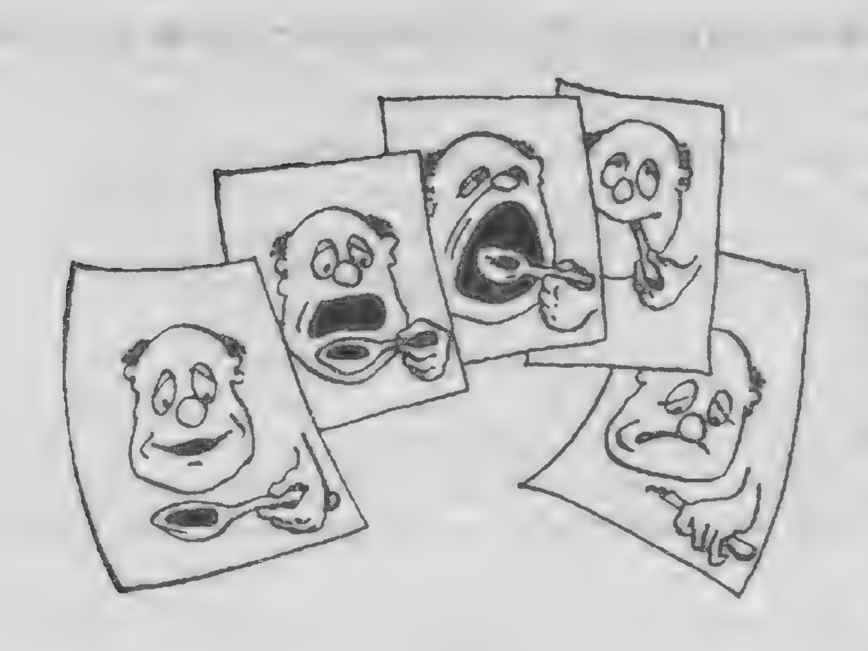
Некоторые из параметров совпадают с параметрами операторов GET и PUT, другие напоминают параметры SPRINT (см. стр. 208). Первый параметр (z) указывает адрес буфера. Параметры x и y задают координаты размещения верхнего левого угла кадра на экране. Координаты указываются не в знакоместах, как для предшествующего оператора GET, а в пикселях (за начало координат принимается верхний левый угол экрана). Следующие два параметра оператора SPUT (u и v) определяют масштабы увеличения выводимого изображения, соответственно, по горизонтали и вертикали. Наконец, параметры g и h, как и в операторах GET и PUT, определяют размер (высоту и длину) исходного кадра. Но в операторе SPUT они задаются необычным образом: в то время как g указывается в знакоместах, h задается в пикселях, то есть в 8 раз превышает значение, подставленное в предшествующий оператор GET. Оператор SPUT всегда выводит изображение в постоянных атрибутах, то есть так, как если бы в операторе PUT использовался режим m=4.

СПРАЙТЫ

SPROFF, SPRON

Существенным преимуществом системы управления спрайтами MegaBasic по сравнению, скажем, со специализированным пакетом Laser Basic является обеспечение независимости движения спрайтов от работы основной программы. Спрайты обрабатываются в фоновом режиме (с использованием прерываний), а основная программа продолжает выполняться своим чередом.

MegaBasic способен передвигать одновременно восемь спрайтов. Кроме того, каждый спрайт может содержать набор *фазовых картинок* или просто *фаз* (до 255), которые в процессе движения



спрайта чередуются, создавая эффект мультипликации. Это еще одна особенность системы управления спрайтами в MegaBasic. Конечно, возможность одновременной обработки столь большого количества спрайтов и их фаз чисто теоретическая, поскольку для размещения такой уймы картинок потребовалось бы 72 килобайта свободной памяти. Но, по крайней мере, один-два «полных» (то есть включающих 255 фаз) спрайта, занимающих по 9 килобайт, запустить можно. Замечено, что не все Spectrum-совместимые машины отечественной сборки успешно справляются с одновременной обработкой большого (свыше четырех) числа спрайтов, независимо от числа фаз.

Управление спрайтами происходит через ячейки памяти, называемые *системными переменными спрайтов*. Записывая в них требуемые значения, можно задавать направление и скорость движения спрайтов, менять количество фаз и частоту их смены и т. д. Перемещение спрайтов происходит дискретно. Направление и скорость движения задается величиной *шага* (в пикселях) и частотой следования шагов. Движение спрайтов и смена их фаз — два не зависимых друг от друга и от основной программы процесса.

По замыслу создателя MegaBasic Майка Лимана, в один пакет с ним должна входить программа **Sprite Designer**, формирующая не только изображения спрайтов, но и, что особенно ценно, файл системных переменных спрайтов. К сожалению, несмотря на широкое распространение различных версий MegaBasic, у нас в стране обнаружилось пренебрежение этой программой. Поэтому пришлось отказаться от запланированного описания работы со Sprite Designer. Тем не менее, той информации, которая приводится ниже, а также примера в конце этого раздела, будет вполне достаточно для овладения довольно сложной системой управления спрайтами в MegaBasic.

MegaBasic предусматривает использование спрайтов только одного размера: 16×16 пикселей, то есть 2×2 знакоместа. Это, конечно, минус по сравнению с Laser Basic. Однако при необходимости можно попытаться параллельно запустить несколько спрайтов, составляющих части одной картинки.

Нетрудно подсчитать, что изображение спрайта, точнее, одной его фазы, размером 2×2 знакоместа займет в памяти, включая атрибуты, 36 байт. Объем памяти, необходимый для размещения всех нужных спрайтов, также легко вычисляется умножением общего числа фаз всех спрайтов на 36. Соответствующее место необходимо зарезервировать в памяти с помощью оператора CLEAR.

Каким же образом можно создавать спрайты, не располагая программой Sprite Designer? Для начала нужно нарисовать все фазы одного спрайта в каком-либо графическом редакторе и записать экранный файл. На экране помещаются 192 фазы, этого в большинстве случаев вполне достаточно. Если мы располагаем картинкой с изображениями фаз, то дальнейшее превращение ее в полноценный спрайт, который может обрабатываться MegaBasic, не составит труда. Это позволяет сделать оператор GET (см. стр. 212). Он предназначен для записи в память изображения кадра, но применим и для записи спрайтов, поскольку формат хранения в памяти кадров и спрайтов совершенно одинаков.

Следующая программа поясняет технологию формирования спрайтов на примере двух спрайтов, каждый из которых состоит из 48 фаз*:

```
10 CLEAR 41535: FOR M=0 TO 5: FOR N=1 TO 16
20 PRINT AT M*2,N*4-4; N+16*M
30 NEXT N: NEXT M: REM Формирование экранного файла
40 DIM Q(96,3)
50 FOR L=1 TO 96: LET Q(L,1)=2*(L-1): IF Q(L,1)>=32 THEN LET
   Q(L,1)=Q(L,1)-32*INT ((L-1)/16): REM Гориз. координата
60 LET Q(L,2)=2*INT ((L-1)/16): REM Верт. координата
70 LET Q(L,3)=41500+36*L: REM Адрес размещения фазы
80 GET_0,Q(L,3),Q(L,2),Q(L,1),2,2: REM Запись фазы в память
90 NEXT L
100 SAVE "SPRITES" CODE 41536,96*36: REM Сохранение спрайтов
```

В этом примере сначала формируется экранный файл с изображениями фаз спрайтов, представляющими собой пронумерованные квадратики (для спрайта 0 — с номера 1 по 48, для спрайта 1 — с номера 49 по 96). Затем фазовые картинки оператором GET последовательно снимаются с экрана и помещаются в память.

MegaBasic располагает двумя операторами, которые управляют спрайтами: SPRON и SPROFF (хотя достаточно только первого, как выяснится из дальнейшего описания).

SPRON_s,m

запускает движение спрайта. Параметр s указывает номер спрайта (от 0 до 7), m — принцип вывода (m=1 — изображение спрайта накладывается на содержимое экрана по принципу OR; m=2 и более — по принципу XOR). Попытка запуска спрайта с номером, выходящим за пределы 0...7, чревата сбоем программы.

Как уже было упомянуто, спрайты в MegaBasic движутся независимо от выполнения основной программы. То есть запущенный оператором SPRON спрайт будет продолжать движение, предписанное ему системными переменными спрайта (о них ниже), одновременно с работой операторов, следующих за SPRON. Остановка произойдет только на время выполнения операций с магнитофоном. Значения системных переменных можно корректи-

* При необходимости нетрудно написать и специальную программу для формирования спрайтов, аналогичную программе Spriter, приведенной для Laser Basic.

ровать, на что мгновенно будет реагировать спрайт (менять скорость, направление движения и т. д.).

Выполнение оператора

SPROFF_s

остановит движение спрайта с номером s. Естественно, все спрайты остановятся и с прекращением выполнения программы.

Инструкция SPRON_s,0 также прекращает вывод спрайта с номером s, и, следовательно, SPROFF можно считать лишним. Оператор SPRON_s,255 прекращает вывод не только спрайта s, но и всех спрайтов с номерами, большими s. SPRON_0,255 вообще останавливает работу со спрайтами.

Но, как уже говорились, для оперирования спрайтами недостаточно иметь их изображения и использовать операторы SPRON и SPROFF. Необходимо еще должным образом установить системные переменные спрайтов.

С каждым спрайтом связана своя область системных переменных размером 18 байт внутри тела самого MegaBasic. Эта область начинается с адреса 56750 для спрайта с номером 0 и простирается до 56893 для спрайта 7.

Приведем назначение и адреса системных переменных относительно адреса a начала области переменных каждого спрайта:

a+0 – значение параметра m, фигурирующего в операторе SPRON и обозначающего способ вывода спрайта s. Выполнение оператора POKE a,m аналогично действию SPRON_s,m;

a+1 – текущая горизонтальная координата (от 0 до 255) позиции вывода спрайта;

a+2 – текущая вертикальная координата (от 0 до 175). Занесение по этому адресу значения, большего 175, может привести к сбою. Выполнив POKE a+1,x: POKE a+2,y, можно задать начальную позицию вывода спрайта;

a+3
a+4 – размеры шага (в пикселях) при перемещении спрайта, соответственно, по координатам *x* и *y*, то есть значения, на которые увеличивается содержимое переменных a+1 и a+2 за один шаг. Если ячейки a+3 и a+4 содержат нули, спрайт будет стоять на месте, только меняя фазы. Заносимые в ячейки a+3 и a+4 числа от 0 до 127 придают спрайту движение, соответственно, слева направо и сверху вниз. Числа 128...255 — движение в обратном направлении (255 — минимальная скорость, 128 — максимальная). Не рекомендуется использовать максимальные значения переменных a+3 и a+4, лучше ограничиваться приращениями не более 30 пикселей;

a+5 – частота шагов. Чем больше содержимое этой ячейки, тем быстрее движется спрайт (при прочих равных условиях);

a+6 – внутренняя системная переменная, связанная с шагом;

a+7 – число фаз спрайта (до 255). Значение должно точно отвечать замыслу, иначе возможен пропуск фаз или вывод непредусмотренных изображений;

a+8 – текущая переменная, указывающая номер выводимой в данный момент фазы;

a+9 — частота смены фаз. Чем больше содержимое этой ячейки, тем быстрее меняются фазы;

a+10 — внутренняя системная переменная, связанная с частотой смены фаз;

a+11
a+12 — двухбайтовое число, указывающее адрес начала размещения в памяти первой фазы спрайта;

a+13 — внутренняя системная переменная;

a+14 — внутренняя системная переменная;

a+15 — числовое значение атрибута, которым закрашивается след, оставляемый спрайтом;

a+16 — не используется;

a+17 — конец области системных переменных данного спрайта, всегда содержит единицу.

После подбора параметров движения спрайтов, можно сохранить системные переменные на ленте оператором

```
SAVE "SPR.sys"CODE 56750,144
```

Выяснив назначение системных переменных, попробуем запустить два спрайта, сформированные программой, приведенной на стр. 216. Добавим в нее следующие строки и запустим*:

```
200 LET s0=41536: LET s1=43264: REM Адреса размещения спрайтов
210 LET a0=56750: LET a1=56750+18: REM Адреса начала системных
    переменных спрайтов
220 POKE a0+7,48: REM Задание числа фаз спрайта 0
230 POKE a0+15,6
240 FOR n=0 TO 17: POKE a1+n, PEEK (a0+n): NEXT n
250 DOKE_a0+11,s0: REM Задание адреса спрайта 0
260 DOKE_a1+11,s1: REM Задание адреса спрайта 1
270 POKE a1+1,0: POKE a1+2,40
280 CLS: SPRON_0,2: SPRON_1,2: PAUSE 0: REM Запуск спрайтов
290 SPRON_0,255:SPRON_1,255: REM Прекращение вывода спрайтов
```

Для спрайта с номером 0 почти все системные переменные устанавливаются при инициализации MegaBasic. Поэтому для начала работы с этим спрайтом достаточно задать значения только трех переменных. Это переменная a+7 (количество фаз в спрайте, в данном случае — 48 для обоих спрайтов), двухбайтовая переменная a+11, a+12 (адрес первой фазы спрайта) и переменная a+15 (атрибут закраски следа от спрайта). Значения системных переменных спрайта номер 1 копируются из области переменных спрайта 0 (строка 240). Затем в строке 260 задается адрес спрайта 1, а в строке 270 — его начальные координаты.

---

* В программе использован оператор MegaBasic DOKE — двухбайтовый аналог POKE (см. стр. 226).

СКРОЛЛИНГ

SCROLL, PAN, SCROLLW, PANW

Для того чтобы закончить описание средств MegaBasic, служащих для обработки изображений, кратко расскажем об операторах скроллинга SCROLL, SCROLLW, PAN и PANW.

Все четыре названных оператора прокручивают изображение в текущем окне: SCROLL и SCROLLW по вертикали, а PAN и PANW по горизонтали. Операторы SCROLL и PAN реализуют обычный скроллинг, в результате выполнения которого изображение, ушедшее за пределы окна, теряется. Операторы SCROLLW и PANW при смещении изображения возвращают его ушедшую часть с противоположной стороны окна, то есть обеспечивают циклический скроллинг.

В операторах

SCROLL_m,y
PAN_m,x

параметр m определяет способ закрашивания освобождающегося при скроллинге поля. Если значение m четно, то закрашивание осуществляется цветом фона, установленным для данного окна, нечетно — цветом тона. Параметры x и y задают величину и направление смещения изображения в пикселях по соответствующей координате.

SCROLLW_y
PANW_x

имеют по одному параметру, которые по смыслу совпадают со вторыми параметрами операторов SCROLL и PAN.

Значения параметров x и y могут быть произвольными, используется только остаток от их деления на 256. Максимальное смещение происходит при значениях x или y, равных нулю, минимальное — когда x или y равны единице. При положительных значениях параметров скроллинг идет вверх и вправо, при отрицательном — вниз и влево.

Применение операторов скроллинга достаточно очевидно из примера:

```
 10 PAPER 5: INK 1: CLS
100 CURRENT_4:WINDOW_6,18,12,28: PAPER 6: INK 2:MODE_4
105 CLW_1: PRINT "SCROLL "
110 PAUSE 10:SCROLL_1,-80: REM Обычный верт. скроллинг
120 CLW_1: PRINT "SCROLLW"
130 PAUSE 10:SCROLLW_128:SCROLLW_-128: REM Циклический
    верт. скроллинг
140 CLW_1: PRINT "PAN "
150 PAUSE 10:PAN_0,128: REM Обычный гориз. скроллинг
160 CLW_1: PRINT "PANW ": REM Циклический гориз. скроллинг
170 PAUSE 10:PANW_128:PANW_-128
180 STOP
```

ЗВУК

PLAY, SOUND, SON, SREP, SOFF

Дополнительные звуковые возможности MegaBasic реализуются двумя операторами: PLAY и SOUND. Совместно с SOUND работают операторы SON, SREP и SOFF, играющие роль переключателей.

Оператор PLAY воспроизводит заданные звуки непосредственно в момент своего выполнения интерпретатором MegaBasic. При этом, естественно, выполнение каких-либо других операторов возможно лишь после завершения действий, предписанных PLAY (аналогично оператору BEEP Spectrum-Бейсика).

Иначе работает оператор SOUND. Он лишь закладывает в специальный «звуковой буфер» данные, необходимые для воспроизведения звука, а «спусковым крючком» служит оператор SON, запускающий отработку данных. В этом случае на фоне воспроизводимого звука возможно продолжение хода программы (работа в среде так называемого «генератора звука на прерываниях»).

Прежде чем рассматривать назначение параметров, входящих в операторы PLAY и SOUND, сделаем несколько общих оговорок. Во-первых, высота звука указывается не в герцах или нотах той или иной октавы, а в условных единицах в интервале от 1 до 256. При выходе значений за эти пределы учитывается только остаток от деления на 256. Во-вторых, длительность звучания ноты задается не в единицах времени, а просто количеством звуковых колебаний, поэтому при прочих равных условиях более высокие звуки оказываются короче.

Отдельно работающие операторы PLAY и SOUND не позволяют составлять музыкальные фразы, основное их назначение — генерация разнообразных звуковых эффектов.

Операторы PLAY и SOUND используют по пять параметров:

PLAY_m,s,f,r,q[,r,q...]
SOUND_a,m,q,r,n

Сначала рассмотрим параметры, которые в обоих операторах имеют одинаковый смысл, и, естественно, обозначены одинаковыми буквами, хотя они и стоят на разных позициях:

m – «качество» звука: 0 — чистый тон, ненулевое значение — шум;

r – число шагов изменения частоты в пределах одного цикла звучания. В операторе PLAY значение этого параметра может быть сколь угодно большим, а в операторе SOUND только от 1 до 256. Поскольку действие оператора PLAY не удается прекратить даже командами MegaBasic ESCAPE (**Space/E**) и RESET (**Space/R**), то в случае «перебора» значения параметра r в операторе PLAY проще перезагрузить MegaBasic, чем дожидаться окончания трелей;

q – смещение частоты от шага к шагу в условных единицах (1...256). Значения параметра от 1 до 127 соответствуют

повышению частоты, а от 129 до 256 — понижению. При q=0 или q=128 высота звука не меняется.

Можно помещать в один оператор PLAY несколько пар параметров r и q с разными значениями, что позволяет разнообразить звук. Например:

```
PLAY_0,10,1000,10,10,20,20
```

Следующие два параметра используются только в операторе PLAY:

s – темп звучания. Ни в одной из версий MegaBasic не удалось заметить, что значение этого параметра как-то влияет на работу PLAY;

f – начальная частота звука (1...256)*.

В операторе SOUND тоже имеются два параметра, которым нет аналогов в PLAY:

a – способ заполнения звукового буфера: при a=0 данные в буфере заменяются, а при ненулевом значении параметра добавляются к содержимому буфера. Звуковой буфер оператора SOUND довольно емкий, но не «резиновый». При его переполнении возможен сбой программы, поэтому следует помнить, что SOUND с нулевым параметром a очищает буфер;

n – число повторений (1...256) циклов, заданных параметром r.

Как уже говорилось, генерация звука, «заложенного» в буфер оператором SOUND, запускается оператором

SON

Вспомогательный оператор

SREP_m

обычно «ходит в паре» с оператором SON. Инструкция SREP_0 задает однократное воспроизведение последовательности звуков из буфера, а ненулевое значение m приводит к циклическому повторению звукового эффекта, прервать которое можно только оператором

SOFF

Естественно, звук прерывается также при остановке программы.

Использование всех пяти операторов MegaBasic, управляющих звуком, иллюстрируется следующим примером:

```
  10 LET m=0
  15 GO SUB 2000
  20 SON: PAUSE 50: GO SUB 1000
  30 SOFF: LET m=NOT m: GO TO 15
1000 PLAY_0,10,1000,10,10: RETURN: REM Музыка
2000 SOUND_1,m,100,5,5:SREP_1: RETURN: REM Шум
```

* Как ни странно, но в операторе SOUND не задается начальная частота звука, что, впрочем, компенсируется очень быстрым развитием звукового эффекта.

СТРУКТУРНОЕ ПРОГРАММИРОВАНИЕ

ENDPROC, BRANCH, MTASK, PCLEAR, REPEAT, UNTIL, POP, PUSH, RESTART, BROFF, BRON

Название этого раздела — «Структурное программирование», несколько условно, поскольку его рамки шире. Здесь собраны все средства MegaBasic, так или иначе служащие для придания программам внутреннего совершенства (впрочем, не всегда явно обнаруживаемого в процессе их выполнения).

Как почти все диалекты Бейсика для ZX Spectrum, MegaBasic располагает инструментом процедур (см. также стр. 176, 260). Правда, это не совсем «честные» процедуры, поскольку параметры их не являются локальными. Из этого следует, что использование одних и тех же переменных в процедуре и в основной программе может привести к недоразумениям.

Для оформления какого-либо блока программы в виде процедуры необходимо задать ее начало (заголовок) и конец. Это делается посредством двух операторов:

@<имя программы>_[<список формальных параметров>]

в начале процедуры* и

ENDPROC_<имя процедуры>

в конце.

Имя процедуры в операторе ENDPROC в случаях, исключающих двусмысленность, может опускаться.

В заголовке процедуры после символа подчеркивания приводится перечень формальных параметров, с которыми надлежит работать процедуре. Ими могут быть числовые и строковые переменные.

Вызов процедуры может осуществляться только из основной программы** указанием имени процедуры и, через символ подчеркивания, списка фактических параметров, то есть значений всех переменных, упомянутых в заголовке процедуры:

<имя процедуры>_<список фактических параметров>

Фактические параметры могут быть только конкретными значениями (не переменными).

Имя процедуры может иметь произвольную длину и состоять из любых символов (кроме псевдографики, UDG и пробелов), но начинаться должно обязательно с буквы. Длина имени ограничена лишь представлениями программиста об удобстве. В качестве имен процедур нельзя использовать ключевые слова MegaBasic и Spectrum-Бейсика.

Приведем пример использования процедур:

```
 5 CLS
10 Z_5,RND*16,RND*12,52,1,12,5,1,"Первый прогон": REM Вызов
     процедуры
```

---

* @ — ASCII-символ с кодом 64.

** Попытка вызова процедуры в виде прямой команды чревата неприятностями.

```
30 Z_6,RND*16,RND*20,44,2,14,6,2,"Second run"
40 GO TO 10
1000 @Z_W,C,L,G,F,S,PP,II,A$: REM Заголовок процедуры
1010 CURRENT_W:WINDOW_C,L,8,G:MODE_4:FONT_F:STIPPLE_S
1020 BRIGHT 1: PAPER PP: INK II:CLW_W,0
1040 PRINT A$: PAUSE 40
1050 SCROLL_1,-64
1100 ENDPROC: REM Конец процедуры
```

Здесь процедура Z имеет 9 параметров и выполняется циклически, причем два параметра, определяющие положение окна, каждый раз меняют свои значения случайным образом.

Новые возможности в оперировании подпрограммами открывают два схожих по действию оператора BRANCH и MTASK.

Оператор

BRANCH_n

после каждой следующей за ним строки программы навязывает выполнение подпрограммы, расположенной со строки с номером n. Выглядит это так, как будто к каждой строке программы довесили через двоеточие оператор GO SUB n. Только подпрограмма, на которую ссылается BRANCH, должна завершаться оператором ENDPROC (без имени процедуры), а не RETURN. Прекращается это построчное «прыгание» после появления в программе оператора BRANCH_0. Использование оператора BRANCH проиллюстрировано в программе, которая рассчитывает длину магнитной ленты, необходимую для заданного времени записи:

```
20 CLS
200 CURRENT_6:WINDOW_4,0,8,30: PAPER 5: INK 1:CLW_0
205 FONT_1:MODE_2
210 PRINT INK 1;"ВРЕМЯ" ' "ВОСПРОИЗВЕДЕНИЯ":MODE_3
215 CURRENT_7:WINDOW_4,34,8,30: PAPER 6:CLW_0
220 FONT_1:MODE_2: PRINT INK 4;"ДЛИНА ЛЕНТЫ":MODE_3
230 BRANCH_400: INPUT "МИНУТ ";M;"СЕКУНД";S: LET A$=STR$
    M+" ' "+STR$ S+" " " ":CURRENT_6
240 LET T=M*60+S: LET L=T*.1905: LET A$=STR$ INT
    L+" м":CURRENT_7: GO TO 230
400 PRINT AT 3,8; INK 2;A$;"   "
410 ENDPROC
```

Оператор

MTASK_n



после каждой строки основной программы отсылает компьютер к подпрограмме (начало подпрограммы задается параметром n). Но при каждой отсылке выполняется не вся подпрограмма, а лишь одна очередная ее строка. Получается эф-

фект параллельного выполнения двух программ (основной и подпрограммы): строка из программы — строка из подпрограммы, и так до появления оператора MTASK_0.

Операторы BRANCH и MTASK являются взаимоисключающими, иначе говоря, не могут работать одновременно. Причем MTASK имеет более высокий приоритет, то есть работать будет именно он при попытке одновременного запуска обоих операторов.

В сочетании с процедурами бывает необходимо использовать еще один оператор:

PCLEAR

Для того чтобы объяснить его назначение, нужно сначала сказать о внутреннем стеке MegaBasic. Стек обслуживает ветвление программ средствами MegaBasic: при входе в процедуры, при выполнении операторов BRANCH и MTASK, а также при выполнении цикла REPEAT...UNTIL (о нем — дальше). В стеке запоминаются номер строки и номер оператора в ней, к выполнению которого нужно вернуться после завершения процедуры, подпрограммы или цикла. После возврата по стеку адрес возврата (значения номеров строки и оператора) снимается с него (стирается). Всего в стек помещается до 10 адресов возврата, и, следовательно, имеется возможность использовать до 10 вложенных процедур или циклов. Если программа структурирована нечетко, возможен останов ее с выдачей сообщений о переполнении стека PROC stack overflow или о его опустошении — PROC stack underflow. Как раз для того, чтобы избежать переполнения стека, и нужен оператор PCLEAR (без параметров), выполняющий его очистку.

Теперь обещанный рассказ о цикле REPEAT...UNTIL. Начало цикла задает оператор

REPEAT

— который используется без параметров. Оператор

UNTIL_r

ограничивает цикл снизу и определяет условие его завершения. Таким образом, выход из цикла происходит, когда значение параметра r становится ненулевым. После этого выполнение программы продолжается со следующей после UNTIL_r строки. Оператор UNTIL удобно использовать совместно с логическими операторами, что и показано в примере программы, которая экспериментирует с различными, в том числе и неразрешенными размерами окон:

```
10 CURRENT_6:WINDOW_0,36,12,28: PAPER 5: INK 1:MODE_6,4:
   STIPPLE_12:FONT_1:PCLEAR
20 REPEAT
30 LET n=INT (RND*23): LET c=INT (RND*63): LET h=INT (RND*10):
   LET g=INT (RND*40)
40 PAUSE 30
50 LET f=(n+h)>23: LET w=(c+g)>63: LET w=h>1: LET s=g>1
60 IF f OR w THEN POP: PUSH_1,100: LET w=0
```

```
70 UNTIL_w AND s
80 CLW_6,0:CURRENT_5:WINDOW_n,c,h,g: PAPER RND*7:
   INK 9:MODE_1:CLW_5,0: PRINT "НОРМА!"
90 RUN
100 PRINT "ОКНО" ' "СЛИШКОМ БОЛЬШОЕ":RUN
```

MegaBasic позволяет при выполнении оператора UNTIL осуществлять переход не только на начало цикла, но и в любое другое место программы (этот прием использован в примере). Для этого надо снять с вершины стека адрес перехода, записанный туда оператором REPEAT, и занести в стек требуемые номера строки и оператора в ней. Очистку вершины стека выполняет «беспараметровый» оператор

POP

А занесение в стек нового адреса возврата — оператор

PUSH_s,n

В нем параметр n задает номер строки, s — номер оператора в ней, куда должен быть осуществлен переход.

Отметим, что в случае «нормального» выхода из цикла, то есть при выполнении условия, записанного в UNTIL, управление будет передано следующему за UNTIL оператору, независимо от адреса перехода, занесенного в стек оператором PUSH.

В MegaBasic имеется оператор

RESTART_n

– который при сбое программы вместо выдачи сообщения об ошибке отсылает компьютер к строке с номером n. К сожалению, RESTART реагирует предусмотренным образом только на ошибки, характерные для Spectrum-Бейсика, а при ошибках выполнения операторов MegaBasic по-прежнему выдает сообщения. Тем не менее, с учетом этого ограничения, RESTART вполне можно использовать, например, для возврата из любого места программы в главное меню по нажатию клавиши Break, которое приравнивается Spectrum-Бейсиком к ошибке.

Применение оператора RESTART проиллюстрировано примером, где в строку 20 умышленно введена возможность периодического возникновения ошибки (недопустимое значение параметра в операторе PAPER):

```
  10 RESTART_1000:BROFF
  20 CURRENT_6: PAPER RND*10: BRIGHT 1: INK 9
  30 CLS : PAUSE 30
  40 GO TO 20
1000 FONT_1:MODE_4:STIPPLE_12
1010 PRINT AT 10,20;"ОШИБКА!"
1020 PAUSE 100: GO TO 20
```

Отмена действия оператора RESTART осуществляется в форме RESTART_OFF. Кстати, в этом примере использован также оператор противоположного свойства

BROFF

Он предотвращает реакцию на клавишу **Break**. В пример он включен для того, чтобы сообщение в центре экрана появлялось только в результате ошибки, а не нажатия **Break**. Действие оператора BROFF отменяется оператором

BRON

Иногда рекомендуют использовать оператор BROFF для защиты программ на MegaBasic от несанкционированного доступа. Если учесть, что команды ESCAPE и RESTART, о которых мы говорили в начале главы, исправно действуют и при отключении клавиши **Break**, то от тех любопытных, кто знаком с MegaBasic, такая защита более чем иллюзорна. От тех же, чьи познания ограничены Spectrum-Бейсиком, лучшей защитой будет сам текст программы, написанный на MegaBasic.

РАБОТА С МАШИННЫМИ КОДАМИ

CALL, DOKE, MON

Теперь расскажем о средствах, которые предоставляет MegaBasic для отладки программ в машинных кодах и стыковки их с бейсик-программами.

Для запуска программы в машинных кодах MegaBasic имеет оператор

CALL_z

– где z — стартовый адрес программы. Он вполне заменяет собой сочетания операторов Spectrum-Бейсика RANDOMIZE USR z, PRINT USR z и пр., а в силу своей краткости более удобен в среде MegaBasic, где ввод операторов осуществляется побуквенно.

Весьма практичен при отладке программ в машинных кодах оператор

DOKE_z,r

Он представляет собой не что иное, как двухбайтовый POKE. Параметры: z — адрес, r — двухбайтовое число, заносимое в ячейки с адресами z и z+1.

Однако наиболее мощным средством отладки программ в машинных кодах является встроенный в MegaBasic *монитор-отладчик**, представляющий собой вполне самостоятельную

* Монитором называется программа, предоставляющая возможность просмотреть содержание ячеек памяти и регистров процессора, записать в них новые значения, отладить программу в машинных кодах и пр.

системную программу. Предусмотрены два способа вызова монитора: или из редактора MegaBasic командой

MON

– или в ходе выполнения программы сочетанием клавиш **Space/F**. При этом распахнется окно номер 3 (или окно с номером w, если ранее был выполнен оператор FX_3,w, переопределивший окно для монитора), представив фронтальную панель монитора (рис. 20).



Будьте готовы к тому, что монитор работает в шестнадцатеричной системе счисления.

Слева и вверху панели монитора представлены названия регистров процессора и их текущие значения. В правой части панели отражается фрагмент памяти: адрес ячейки, ее содержимое и соответствующий символ. Адрес ячейки памяти, на которую в данный момент настроен монитор (*текущей ячейки*), выделен инверсным цветом. В нижней строке монитора расположена строка, отмеченная символом >. Она предназначена для ввода и редактирования команд монитора.

Содержимое текущей ячейки изменяется набором в командной строке нового значения и нажатием клавиши **Enter**. После этого текущей автоматически становится следующая ячейка. Переход к следующему адресу без изменения содержимого ячейки осуществляется простым нажатием **Enter**, а откат на одну ячейку назад — нажатием клавиш **SS/J**. Установить текущей произвольную ячейку памяти можно с помощью команды

M <адрес>

Монитор позволяет также изменять содержимое регистров. Делается это командой

R <двухбайтовое число>

Регистр, в который заносится значение, отмечен звездочкой (*). Перемещается звездочка, и, соответственно, меняется текущий регистр с помощью клавиши **P**.

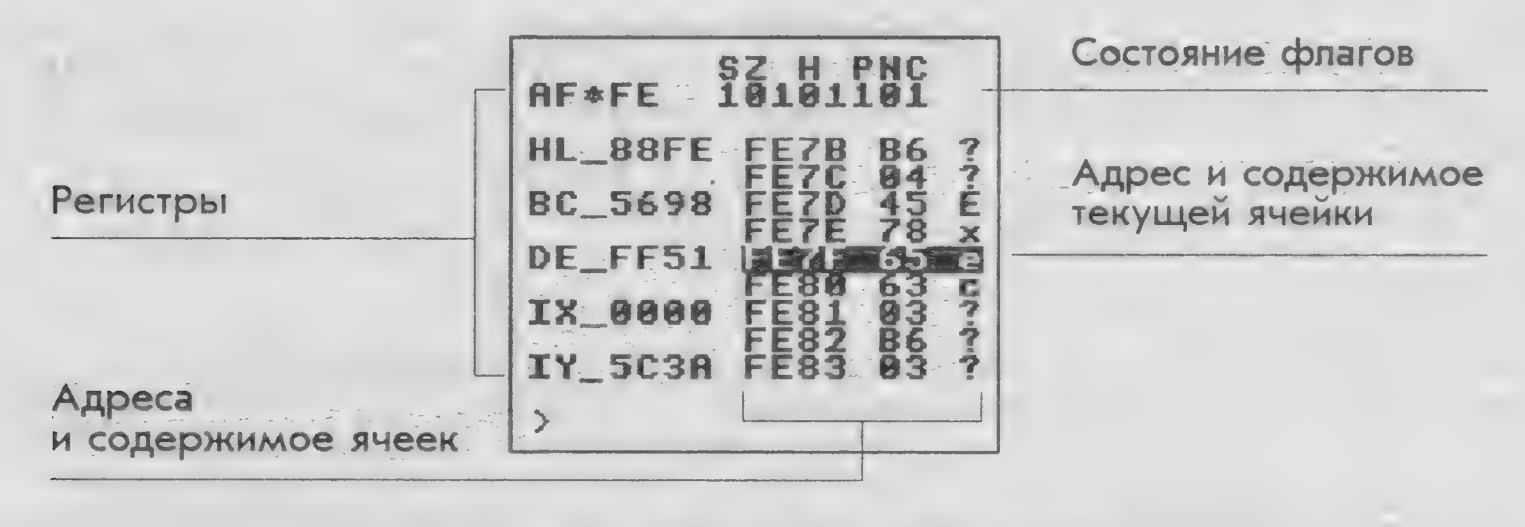


Рис. 20. Фронтальная панель монитора.

Программа в машинном коде запускается командой

J <адрес>

Монитором предусмотрена возможность расстановки так называемых точек останова, что позволяет осуществлять отладку программ. Для указания места останова адрес, по которому нужно «тормознуть» выполнение программы, устанавливается текущим и нажимается клавиша **S**. Отменить точку останова можно клавишей **U**. После останова выполнение программы продолжается при нажатии клавиши **K**.

С помощью специальных команд монитора можно скопировать блок памяти по произвольному адресу (команда **L**) или занести во все ячейки выделенного блока памяти какое-либо значение (команда **I**). Формат этих команд:

L <адрес начала блока> <адрес копирования> <длина блока>

I <адрес начала блока> <длина блока> <байт-заполнитель>

Выход из монитора происходит по нажатию **Space**.

Мы не стали здесь детально описывать приемы работы с монитором-отладчиком, поскольку эта книга посвящена Бейсику, а не программированию в машинных кодах*.

ПРОЧИЕ ОПЕРАТОРЫ И КОМАНДЫ

TRON, SPEED, TROFF, DEFG, KEY, PRINTER, BACKUP, EXAMINE

В MegaBasic существует группа операторов трассировки, служащая для отладки бейсик-программ. Эти операторы можно использовать как в тексте программы, так и в непосредственном режиме. «Главным» из них является оператор

TRON

Он включает режим замедленного выполнения программы (трассировки). При этом в левом нижнем углу экрана высвечивается номер выполняемой в данный момент строки (номер оператора в строке не индицируется). Кстати, когда трассировка происходит при работающем операторе MTASK, номера строк, относящиеся к программе и подпрограмме, выделяются разными шрифтами.

Благодаря наличию оператора

SPEED_v

имеется возможность задавать скорость трассировки. При v=0 — скорость максимальная, при v=254 — минимальная. Если v=255,

* Более подробно разобраться с монитором-отладчиком поможет книга [1], где описан монитор MONS4.

программа выполняется в пошаговом режиме: по одному оператору при каждом нажатии любой клавиши.

Отменяется режим трассировки оператором

TROFF

Оператор

DEFG_a$,b1,...,b8

облегчает построение символов, определяемых пользователем (UDG). Строковая переменная **a$** задает клавишу, при нажатии которой в режиме курсора **[G]** будет вызываться определяемый символ UDG, а 8 последующих чисел являются байтами, из которых складывается соответствующий символ. Удобно задавать числа **b1...b8** не в десятичной, а в двоичной форме с использованием оператора **BIN** Spectrum-Бейсика — это придает наглядность процессу формирования символа.

Из оставшихся нерассмотренными операторов особенно интересен

KEY_k,a$

Он позволяет закрепить за клавишами **0...9** произвольные последовательности символов.

Вызов запрограммированной последовательности осуществляется в момент редактирования строки комбинацией клавиш **CS/SS+SS/k**, где **k** задает номер цифровой клавиши (**0...9**).

Сама последовательность — это обычная строка символов или символьная переменная длиною до 255 знаков. Она записывается на место параметра **a$**.

Практически длина последовательности может и превышать 255 знаков, но тогда клавиши нужно задействовать, скажем, через одну. Иначе «хвост» предыдущей серии наедет на область памяти, занятую последовательностью, привязанной к последующей клавише. Для клавиши **9** это недопустимо, так как она последняя. Однако длинные последовательности символов неудобны, поскольку они выводятся достаточно медленно. Лучше использовать несколько более коротких серий.

Если одними и теми же последовательностями символов приходится пользоваться постоянно, имеет смысл сохранить их на ленте в виде кодового файла и подгружать его в MegaBasic по мере необходимости. Область памяти, содержащая информацию о последовательностях, начинается с адреса 59956. Участки памяти, относящиеся к каждой клавише, занимают по 256 байт и должны заканчиваться нулевым байтом. Таким образом, если запрограммирована только одна клавиша **0**, то для сохранения «привязанной» к ней последовательности нужно выполнить оператор

```
SAVE "KEYS"CODE 59956,256
```

Если запрограммированы все десять клавиш, длина файла возрастет до 2560 байт.

Приведем один красивый прием. Если вызываемая клавишей последовательность символов представляет собой строку из одного

или нескольких операторов, то пользуясь встроенным монитором-отладчиком, можно поставить в конце нее (в области памяти, которая начинается с адреса 59956) число 13 (0D в шестнадцатеричном представлении), а вслед за ним ноль. Число 13 является кодом клавиши **Enter** (возврат каретки), поэтому после вызова последовательности нажатием **CS/SS+SS/k** строка операторов появится на экране и тут же автоматически выполнится.

Оператор

PRINTER_i

обеспечивает простой способ переключения потока выводимой информации с экрана на принтер. Чтобы в полной мере оценить действие этого оператора, нужно, как минимум, подключить к компьютеру принтер и, кроме того, занести в ячейки 59934/35 адрес драйвера принтера. Если параметр оператора PRINTER_i равен нулю, то вывод будет осуществляться на экран, в других случаях — на принтер.

Если после работы с принтером забыть выполнить PRINTER_0, то может возникнуть недоразумение — никакой информации на экране больше не будет появляться. Для исправления положения нужно «вслепую» ввести оператор PRINTER_0, и все наладится.

Два последних описываемых оператора MegaBasic относятся к работе с магнитофоном. Один из них

BACKUP

представляет собой весьма несовершенный ленточный копировщик. С его помощью можно копировать лишь файлы четырех стандартных типов (BASIC, CODE, CHARACTER ARRAY и NUMBER ARRAY), причем за длиной файла, чтобы он не наложился на MegaBasic, нужно следить самим. Кроме того, не отслеживаются ошибки при загрузке: в любом случае на ленту сбрасывается файл той длины, которая указана в заголовке. А хуже всего то, что любая бейсик-программа, находящаяся в памяти, после выполнения оператора BACKUP ликвидируется, очищая место для копируемых файлов. Сам MegaBasic после этого обычно становится неработоспособным. Так что если необходимо что-нибудь скопировать, не поленитесь загрузить нормальный копировщик — затраты времени окупятся.

И, наконец, оператор

EXAMINE

поможет прочесть заголовки файлов с магнитофона: название, длину, номер стартовой строки или адрес загрузки кодового блока. В отличие от BACKUP, этот оператор вполне безобиден.

Сказать, что мы дали исчерпывающую информацию о MegaBasic, будет самонадеянно. Это одна из тех «вещей в себе», в которых всегда можно обнаружить что-нибудь новенькое. Поэтому дерзайте — и будете вознаграждены.

ПРИЛОЖЕНИЯ

1. Алфавитный список операторов MegaBasic

Таблица 16.

Оператор	Действие	Стр.
@<имя>_a,...,z$	Отмечает начало (заголовок) процедуры	222
AUTO_n,k	Запускает автоматическую нумерацию строк программы с номера n и с шагом k	204
BACKUP	Включает режим ленточного копировщика	230
BRANCH_n	Инициирует ветвление программы с выполнением подпрограммы, начинающейся со строки n	223
BROFF	Отключает действие клавиши Break	226
BRON	Возобновляет действие клавиши Break	226
CALL_z	Запускает программу в машинных кодах с адреса z	226
CHANGE_a1,a2	Заменяет атрибуты экрана (см. в тексте)	212
CLW_m,w CLW_m	Очищает окно w или текущее окно способом m	210
CURRENT_w	Устанавливает текущим окно w	209
DEFG_a$,b1...b8	Формирует символ UDG из байтов b1...b8 и связывает его с клавишей a$	229
DELETE_n1,n2	Удаляет блок программы со строки n1 до строки n2	204
DOKE_z,b	Заносит двухбайтовое число b в ячейки с адресами z и z+1	226
DOWN_n,c,a$	Выводит строковую переменную a$, начиная со строки n (0...23) и столбца c (0...63)	208
EDIT_n	Вызывает на редактирование строку n	205
ENDPROC_<имя>	Отмечает конец блока процедуры	222
EXAMINE	Считывает заголовки файлов с ленты	230
FADE_a	Циклически декрементирует значения атрибутов экрана до значения a	211
FONT_f	Включает шрифт номер f (0...2)	206
FX_i,w	Переключает вывод потока i (0...6) в окно w	209

Оператор	Действие	Стр.
GET_m,z,n,c,h,g	Переносит в режиме m в буфер по адресу z фрагмент экрана с координатами: n — начальная строка, c — начальный столбец, h — высота, g — ширина	212
INVERT	Инвертирует атрибуты всего экрана	211
KEY_k,a$	Задает строку a$ для вывода клавишами CS/SS+SS/k, где k — цифровая клавиша (0...9)	229
MODE_w,m MODE_m	Устанавливает размер m (1...4) символов для вывода в окно w	207
MON	Вызывает встроенный монитор-отладчик	226
MTASK_n	Инициирует параллельное выполнение подпрограммы, начинающейся со строки n	223
PAN_m,,p	Выполняет горизонтальный скроллинг окна на p (−255...255) пикселей в режиме m (0, 1)	219
PANW_p	Выполняет горизонтальный циклический скроллинг окна на p (−255...255) пикселей	219
PCLEAR	Очищает стек процедур	224
PLAY_,m,s,f,r,q	Воспроизводит музыкальную фразу, параметры: m — режим, s — темп, f — начальная частота, r — число шагов, q — размер шага	220
POP	Выталкивает из стека процедур последний адрес возврата (номер строки и оператора)	225
PRINTER_i	Переключает вывод информации на принтер/экран, i — номер потока	230
PUSH_s,n	Заносит в стек процедур адрес возврата (n — номер строки, s — номер оператора в ней)	225
PUT_m,z,n,c,h,g	Возвращает на экран кадр, ранее помещенный в буфер оператором GET: m — режим, z — адрес буфера, n — начальная строка, c — начальный столбец, h — высота кадра, g — ширина кадра	212
REPEAT	См. UNTIL	224
RESTART_n RESTART_OFF	Инициирует/прекращает переход по ошибке на строку n	225
SCROLLW_p	Выполняет вертикальный скроллинг окна на p пикселей	219
SCROLL_m,p	Выполняет вертикальный циклический скроллинг окна на p пикселей в режиме m	219

Оператор	Действие	Стр.
SOFF	Прекращает работу звукового генератора	221
SON	Запускает звуковой генератор в соответствии с данными, заданными оператором SOUND	221
SOUND_a,m,q,r,n	Помещает данные в буфер звукового генератора: a — способ заполнения буфера, m — режим, q — шаг, r — число шагов, n — число циклов	220
SPEED_s	Задает скорость трассировки s (0...255)	228
SPRINT_x,y,u,v,g,h	Выводит текст увеличенными символами, масштаб и местоположение которых определены параметрами: x — горизонтальная координата (0...255), y — вертикальная координата (0...191), u — увеличение по горизонтали, v — увеличение по вертикали, g — ширина, h — высота	208
SPROFF	Прекращает вывод спрайтов	217
SPRON_s,m	Инициирует вывод спрайта номер s в режиме m (0...255)	216
SPUT_z,x,y,u,v,g,h	Выводит на экран изображение кадра с увеличением: z — адрес буфера, x — горизонтальная координата, y — вертикальная координата, u — увеличение по горизонтали, v — увеличение по вертикали, g — ширина, h — высота	214
SREP_m	Устанавливает однократное циклическое воспроизведение звуков оператором SON	221
STIPPLE_d	Осуществляет штриховку символов в режиме MODE_4 с плотностью d (0...15)	207
SWAP_a2,a1	Заменяет на экране атрибуты a1 на атрибуты a2	211
TROFF	Отменяет режим трассировки	229
TRON	Включает режим трассировки	228
UNTIL_r	Повторяет группу операторов, заключенных между REPEAT и UNTIL, до тех пор, пока значение r не станет нулевым	224
VDU_b1,... bn	Аналог конструкции PRINT CHR$ b1,,... bn, где bn — коды символов	207 209
WINDOW_n,c,h,g	Устанавливает положение и размеры текущего окна: n — начальная строка (0...23), c — начальный столбец (0...63), h (1...24) — высота, g — ширина (1...64)	209

2. Распределение памяти при работе с MegaBasic

P_RAMT (23732)		65535
	Символы, определяемые пользователем	
UDG (23675)		65368
	Интерпретатор MegaBasic	
		62515
	Область программируемых клавиш	
		59956
	Интерпретатор MegaBasic	
		45000
	Свободная область	
RAMTOP (23730)		
	...	
E_LINE (23641)		
	Переменные Бейсика	
VARS (23627)		
	Бейсик-программа	
PROG (23635)		

3. Использование MegaBasic с системой TR-DOS

Поскольку MegaBasic работает во 2-м режиме прерываний процессора (IM2), то на любые попытки обратиться к дисковой операционной системе «в лоб» он реагирует весьма болезненно. Чтобы организовать корректную работу с TR-DOS, необходимо на время обращения к диску переключаться в режим прерываний IM1. Для этого нужно добавить в MegaBasic две подпрограммы в кодах, вписав в бейсик-загрузчик несколько строк:

```
10 CLEAR 44979: REM Опускаем RAMTOP на 10 байт
...
40 RANDOMIZE USR 15619: REM : LOAD "MegaCode"CODE
41 FOR n=44980 TO 44989: READ i: POKE n,i: NEXT n: REM Размещаем подпрограммы
42 DATA 243,237,86,251,201,243,237,94,251,201
...
90 RANDOMIZE USR 56100: REM Запуск MegaBasic
```

Теперь обращаться к диску из MegaBasic можно следующим образом:

```
100 RANDOMIZE USR 44980
110 RANDOMIZE USR 15619: REM: LOAD "filename"
120 RANDOMIZE USR 44985
```

В строке 100 происходит переход в режим IM1, затем могут быть любые операции с диском, а 120-я строка восстанавливает IM2.

Отметим, что этот способ не обеспечивает корректного выхода в TR-DOS с помощью инструкции RANDOMIZE USR 15616.

BETA BASIC

Первая версия Beta Basic (v.1.0) фирмы Betasoft появилась на свет в 1983 г. Она дополнила Spectrum-Бейсик более чем 20 новыми операторами и функциями. Потом была написана версия 1.8 (1984 г.) и, наконец, появились самые мощные версии — 3.0 и 3.1 (1985 г.)*, которые поддерживают более 40 новых операторов, расширяют возможности некоторых «базовых» операторов и вводят около 30 дополнительных функций. Мы будем описывать версию 3.1**.

В основном, Beta Basic ориентирован на решение вычислительных задач и работу с данными и, пожалуй, представляет из себя наиболее мощную версию Бейсика для ZX Spectrum.

Beta Basic имеет развитые средства структурного программирования и управления программой (процедуры, циклы типа DO...LOOP, оператор варианта и т. п.), что заметно упрощает составление программ и сокращает время их разработки. Значительно расширен в нем арсенал средств работы с памятью и реализован принципиально новый подход к обработке массивов и строк символов: их копирование, склейка, сортировка, поиск по образцу в произвольном сечении и т. д.

Beta Basic располагает богатым набором операторов, обслуживающих вывод информации на экран. В нем возможна организация многооконного режима работы программ (до 128 окон), вывод символов произвольного размера без привязки к знакоместам. Все это в сочетании с возможностью изменять масштаб координатной сетки и перемещать центр координат в любое место экрана и даже за его пределы (то есть организовывать работу непосредственно в математических координатах) позволяет удобно оформлять результаты работы программ в виде графиков, гистограмм и пр.

---

* Есть, правда, еще одна версия — 4.0 (1988 г.), предназначенная для ZX Spectrum 128.

** Файлы: Beta 3.1 (тип BASIC, длина 945 байт) и cBeta 3.1 (тип CODE, длина 18577 байт, адрес загрузки 46960).

Несмотря на то, что Beta Basic не специализируется на работе с графикой, он, тем не менее, выполняет многие операции обработки изображения: работу со спрайтами, быструю замену цветовых атрибутов, скроллинг окна экрана, заливку замкнутого контура и т. д.

Beta Basic имеет встроенные часы, средства обработки ошибочных ситуаций и пошаговой отладки программ (трассировки).

Важным достоинством Beta Basic является наличие в нем удобного редактора программ, в котором реализованы автонумерация и перенумерация строк, поиск и замена по образцу, быстрое удаление фрагментов программы и многое другое.

Несмотря на значительное место, занимаемое в памяти интерпретатором Beta Basic (около 19 килобайт), его расширенные возможности с лихвой окупают этот недостаток и позволяют быстро разрабатывать эффективные и компактные программы.

РЕДАКТОР

KEYWORDS, DEF KEY, AUTO, EDIT, REF, ALTER, JOIN, SPLIT, RENUM, DELETE, LIST, MEM()

Beta Basic, в отличие от своих собратьев — расширений Spectrum-Бейсика, в синтаксисе и способе ввода операторов целиком сохраняет верность традициям, принятым в базовой версии: в нем ключевые слова вводятся нажатием лишь одной-двух клавиш. Для реализации этого использован стандартный режим курсора **[G]** (**Graphics**), включаемый комбинацией клавиш **CS/9**. Например, после перехода в режим [G] нажатие клавиши **8** приведет к выводу в строку редактора ключевого слова KEYWORDS*.

Однако сказанное вовсе не означает, что в Beta Basic нельзя использовать псевдографику и символы, определяемые пользователем (UDG). Режиму **Graphics** всегда можно вернуть его старый, привычный смысл. Делается это с помощью уже упомянутой команды KEYWORDS. Она имеет один числовой параметр. KEYWORDS 1 переводит компьютер в режим ключевых слов Beta Basic, а KEYWORDS 0 — в режим вывода псевдографических символов и символов UDG, который отличается от стандартного синклеровского режима **Graphics** только тем, что в нем для удобства за клавишей **8** закреплен не пробел, а ключевое слово KEYWORDS.

После загрузки Beta Basic устанавливается режим KEYWORDS 1.

Кроме синклеровского способа ввода ключевых слов, Beta Basic допускает и посимвольный их ввод, причем безразлично, прописными или строчными буквами (в режимах курсора **[C]** или **[L]**).

---

* Соответствие клавиш ZX Spectrum операторам Beta Basic приведено в Приложении 1.

Редактор Spectrum-Бейсика — большой интеллектуал: он, исходя из контекста, знает, в каком месте вводимой строки должно появиться ключевое слово, и самостоятельно переводит курсор в режим [K]. Избавиться от этого в стандартном Бейсике невозможно. Зато Beta Basic это позволяет. Находясь в режиме ввода ключевых слов [K], стоит нажать клавишу **Space**, и курсор превращается в [C] или [L] (в зависимости от того, какой из этих режимов был установлен ранее). Для того чтобы вернуться к режиму [K], достаточно переместить курсор на прежнее место либо нажать клавиши **SS/Enter**. Таким образом можно выбирать способ набора текста программы: целиком ключевыми словами или посимвольно.

Закрепить способ ввода ключевых слов можно с помощью той же команды KEYWORDS: KEYWORDS 2 отключит возможность посимвольного ввода, KEYWORDS 4, наоборот, позволит набирать все ключевые слова только посимвольно, а KEYWORDS 3 вернет редактор в исходный режим «свободного выбора».

Посимвольно набранные операторы и функции после нажатия **Enter** (даже если ввод строки не завершен) заменяются соответствующими ключевыми словами. Этот эффект проявляется наглядно, если ключевые слова набирать строчными буквами — после нажатия **Enter** они заменятся на прописные. Если же редактор обнаружит ошибку и не захочет вводить строку, найти причину будет легко: неправильно набранное ключевое слово так и останется записанным строчными буквами. Отметим только, что ошибка может быть не только в пропущенной или неправильной букве, но и в отсутствии в нужном месте пробела. К примеру, запись CHR$A редактор не поймет.

Однако возможны ситуации, когда набранное с ошибкой ключевое слово будет интерпретировано как имя переменной и пропущено редактором. Так, например, если вместо строки LET A=COS T набрать LET A=COST, то редактор сочтет это вполне приемлемым и спокойно «проглотит», хотя программа будет работать неправильно.

Все же редактор Beta Basic делает все возможное, чтобы обнаружить и исправить максимум ошибок. Например, если между именем функции (оператором, другим служебным словом) и аргументом отсутствует пробел, но аргумент не является именем переменной (то есть начинается не с буквы), «ошибка» будет исправлена.

Beta Basic значительно (ровно вдвое!) расширяет список функций языка. Разумеется, было бы бессмысленно пытаться каждой новой функции приписать «персональную» клавишу компьютера, и разработчики Beta Basic предложили оригинальный и по-своему удобный метод ввода имен функций. Сразу оговорим, что он не исключает возможность и посимвольного их ввода.

Все новые встроенные функции заданы с использованием инструмента описания функций, определяемых пользователем (DEF FN и FN)*. Соответственно, и обращаться к ним в тексте про-

---

* Если, находясь в Beta Basic, выполнить оператор LIST 0, то можно увидеть, что строка с номером 0 состоит из операторов DEF FN — в ней и определяются все новые функции Beta Basic.

граммы нужно тоже стандартным способом — через ключевое слово FN, добавляя однобуквенное имя функции. Таким образом, определенное число однобуквенных имен функций пользователя Beta Basic занимает под имена встроенных функций.

Для того чтобы в тексте программы встроенные функции отличались от функций, определенных пользователем, применен следующий прием: при наборе конструкции типа FN <имя функции>, где <имя функции> — зарезервированная буква, редактор заменяет эту букву на полное название функции. Например, после ввода открывающей скобки в строке

```
LET a=FN i(
```

сочетание FN i будет заменено одним служебным словом — именем новой функции:

```
LET a=INSTRING(
```

Преобразование в развернутую форму имен *символьных* функций происходит после ввода символа $*.

```
110 FOR i=1 TO 3
120   ON i
        LET a=5088
        LET a=3749
        IF Phone<484 THEN
          LET a=1282, Phone=235
        ELSE
          LET a=19.03, Phone=242
130 NEXT i
```

Рис. 21. Фрагмент листинга программы Beta Basic.

Редактор Beta Basic при выводе листинга на экран обеспечивает контекстное позиционирование программных строк, что делает тексты программ более удобочитаемыми (рис. 21). При этом на экран не выводятся двоеточия, разделяющие расположенные в одной строке операторы — каждый оператор печатается с новой строки**. Это свойство редактора, например, позволяет, набрав несколько двоеточий подряд, отделить друг от друга пустыми строками логически завершенные части программы (процедуры, DATA-списки и т. п.). Того же эффекта можно добиться, нажимая одновременно **CS** и **Enter*****.

В режиме редактирования строки нетрадиционно работают клавиши управления курсором. Если в Spectrum-Бейсике курсор мог двигаться только в горизонтальном направлении, а клавишами «курсор вверх» и «курсор вниз» перемещался указатель текущей строки в листинге программы, то редактор Beta Basic позволяет свободно «гулять» по редактируемой строке в любом из четырех направлений. При попытке переместить курсор выше либо ниже редактируемой строки он перебрасывается в ее конец. Управлять же указателем текущей строки можно только при условии, что строка редактора пуста и курсор находится в режиме **[K]**.

---

* Полный список встроенных функций Beta Basic приведен в Приложении 2.

** Для экономии места мы все же будем приводить листинги программ в привычном виде — разделяя операторы двоеточиями.

*** При этом в текст строки помещается управляющий символ CHR$ 15, который Beta Basic трактует как пробел (CHR$ 32) плюс возврат каретки (CHR$ 13).

Доступен в Beta Basic еще один способ набора программных строк — через программирование клавиатуры. Оператор

DEF KEY <"символ клавиши">[:<последовательность операторов>[:]]

DEF KEY <"символ клавиши">[;<строка символов[:]>]

приписывает любой клавише компьютера (кроме **CS**, **SS**, **Space** и **Enter**) <последовательность операторов> или <строку символов> — макроопределение. Макроопределение может быть задано как последовательностью операторов, и тогда оно вводится после двоеточия, так и символьной строкой, отделяемой от первого параметра оператора точкой с запятой.

После определения макрокоманды оператором DEF KEY она выполняется нажатием указанной клавиши в специальном режиме курсора **[*]**, введенном в дополнение к стандартным режимам курсора. Включается режим **[*]** нажатием клавиш **SS/Space**.

Выполним строку:

```
DEF KEY "P": PRINT "Hello!": PRINT "How do you do?"
```

Теперь, если в режиме курсора **[*]** нажать клавишу **P**, на экране появится:

Hello!
How do you do?

Пример можно переписать и так:

```
10 LET a$="PRINT ""Hello"": PRINT ""How do you do?"""
20 DEF KEY "P";a$
```

Если в конец последовательности, следующей за DEF KEY, поместить двоеточие, то она не будет выполнена, а ее текст выведется в строку редактора. Этим можно пользоваться для помещения в строку редактора часто повторяющихся при наборе программы фрагментов строк.

Макроопределение можно «снять» с клавиши, выполнив оператор DEF KEY с символом данной клавиши без следующего за ним набора инструкций, например, DEF KEY "P". Одновременно все макроопределения удаляются оператором DEF KEY ERASE.

Следует помнить, что, сохраняя в памяти созданные макроопределения, интерпретатор несколько опускает RAMTOP.

Beta Basic избавляет нас от необходимости при вводе программы каждый раз набирать номер очередной строки. Включается режим автонумерации командой

AUTO [<начальный номер>[,<шаг>]]

В ней параметр <начальный номер>* задает номер первой добавляемой к программе строки, а параметр <шаг> — шаг нумерации

---

* Для большей наглядности мы будем указывать в формате операторов и функций Beta Basic «словесное» название параметров. Однако вполне естественно, что, например, параметром <начальный номер> может быть любое целочисленное значение (переменная, выражение), которое попадает в разрешенный диапазон (в данном случае — 0...9999); под <строкой символов> подразумевается символьное значение (символьная константа, переменная, выражение); под <числом> — числовое значение и т. д.

для последующих строк. Команда может не иметь ни одного параметра либо иметь только один. В первом случае по умолчанию предполагается, что оба параметра равны 10. Во втором — единственный параметр определяет номер первой «автонумеруемой» строки, а шаг также равен 10.

Отменить автонумерацию проще всего, нажав и удерживая клавишу **Break** (**CS/Space**) в течение секунды. В момент отключения режима AUTO появится сообщение 0 OK: 0,1. Автонумерация сбросится и по команде AUTO 0 или после выполнения любого оператора в непосредственном режиме (для этого перед вводом оператора надо стереть появившийся в строке редактора очередной номер).

После стирания выскочившего номера можно перемещать указатель текущей строки. Следует только учитывать, что при возврате к автонумерации редактор предложит уже не прежний номер, а новый — на <шаг> больше номера строки, на которую установлен указатель, хотя строка с таким номером уже может существовать.

Продолжим рассмотрение команд редактора Beta Basic.

EDIT [<номер строки>]

Эта команда вызывает на редактирование строку программы с номером, заданным в параметре, независимо от положения указателя текущей строки. Если строки с заданным номером не существует, на редактирование будет вызвана строка с ближайшим большим номером. Набирается эта команда нетрадиционно, но просто: клавишей **0** после любого нажатия **Enter**. Команда EDIT без параметра вызывает на редактирование текущую строку программы. Впрочем, набрать EDIT можно и обычным способом — в режиме курсора **[G]** или посимвольно.

Мощная команда редактора Beta Basic

REF <ключ для поиска>

последовательно вызывает на редактирование строки программы, включающие в себя текст, заданный в параметре. Ключом для поиска может быть имя переменной (массива, процедуры), числовая константа (выражение) или символьная строка, заключенная в кавычки. Различий между прописными и строчными буквами не делается.

Если в качестве ключа необходимо использовать содержимое символьной переменной, а не ее имя, то последнюю необходимо взять в скобки: REF (**a$**).

В случае, если образцом является строка символов, на редактирование будут вызваны строки с любым включением указанного набора символов, в том числе с полностью или частично совпадающими именами переменных. Так, например, команда REF "org" вызовет одну за другой на редактирование строки

```
78 LET Org=55679
80 LET Forgo=22.51
```

```
510 INPUT "Organization number "; Num
640 REM ORGANIC
```

По завершении редактирования очередной найденной строки и ввода ее в программу повторное нажатие клавиши Enter поместит в нижнюю часть экрана следующую строку, удовлетворяюшую условию поиска. Если строка программы содержит искомую подстроку в нескольких экземплярах, она будет «вызвана на ковер» соответствующее количество раз.

Выход из этого режима осуществляется так же, как и из режима AUTO.

С помощью команды

ALTER <ключ для поиска> **TO** <образец для замены>

можно осуществлять автоматический поиск и замену фрагментов текста программы. Для обоих параметров команды справедливы соглашения, принятые для параметра команды REF. Второй параметр может принимать значение «пустая строка».

Команда ALTER просматривает текст программы и все найденные включения последовательностей символов или имен переменных (массивов, процедур), заданные <ключом для поиска>, заменяет на <образец для замены>, указанный во втором параметре.



JOIN [<номер строки>]

Эта команда сцепляет через двоеточие две соседние строки программы: строку с номером, указанным в параметре, и следующую за ней. После выполнения команды номер присоединенной строки освобождается. К примеру, фрагмент программы

```
151 FOR i=1 TO 1000
152 PRINT PEEK i
153 NEXT i
```

после выполнения команды JOIN 151 будет выглядеть так:

```
151 FOR i=1 TO 1000: PRINT PEEK i
153 NEXT i
```

Команда JOIN без параметра соединит текущую строку и строку, следующую за ней.

SPLIT

Эта команда обратна по действию команде JOIN: она расщепляет строки, составленные из нескольких операторов. В отличие от других команд редактора, SPLIT выполняется не введением ключевого слова с параметрами, а совершением несложной операции в процессе редактирования строк. Разделение производится

введением символа <> (клавиши SS/W) в следующую за «фантом-двоеточием»* позицию редактируемой строки (в этой позиции курсор переходит в режим [K]). После нажатия Enter часть строки до курсора возвращается с прежним номером в текст программы, а остаток строки, словно хвост ящерицы, оказывается в наших руках. Оставшийся в редакторе «хвост» имеет тот же номер, что и его удравшая «хозяйка». Поэтому, если номер не сменить, то после нажатия Enter вторая часть строки перепишется в программу на место первой. Будьте внимательны!

Без следующей команды не может обойтись любая мало-мальски уважающая себя версия Бейсика:

RENUM [<начало>] [**TO** <конец>] [**LINE** <нов. начало>] [**STEP** <шаг>]

Она перенумеровывает (перебрасывает с одного места программы в другое) блок строк, начиная с номера, заданного параметром <начало>, по номер <конец>. Новое место (номер, присваиваемый первой строке блока) и шаг перенумерации определяются параметрами <нов. начало> и <шаг>**.

По умолчанию параметр <начало> принимает значение номера первой строки программы, параметр <конец> — номера последней строки, <нов. начало> — 10 и <шаг> — 10. Следовательно, использование команды RENUM без параметров приведет к перенумерации всех строк программы, начиная с номера 10 и с шагом 10.

Если при задании первых двух параметров будет указан номер не существующей в программе строки, действие команды прервется сообщением No such line — нет такой строки. В команде предусмотрена «защита от дурака»: если от нее потребуют присвоить строке уже существующий в программе номер, будет выдано сообщение No room for line (нет места для строки).

Конечно, изменить номера строк в программе — дело нехитрое, куда важнее другое: сохранить при этом правильность всех ссылок в операторах GO TO, GO SUB и других, задающих переходы по номеру строки. В случае, если параметры в этих операторах заданы в явном виде (числом), команда RENUM выполняет перенумерацию этих параметров корректно. Номера строк, в которых содержатся операторы перехода с параметрами, заданными выражением, после отработки команды выводятся на экран в виде сообщений:

Failed at <номер строки>:<номер оператора в строке>

В этих операторах придется вручную откорректировать адреса переходов.

Для работы команды RENUM в качестве буфера используется экранная область памяти компьютера. Поэтому не пугайтесь, если

---

* Напомним, что в Beta Basic двоеточия не высвечиваются.

** Beta Basic позволяет при перенумерации присваивать строкам номера в интервале 1...16383. При этом все переходы на строки с номерами, превышающими 9999, осуществляются корректно. Вызов на редактирование таких строк возможен всеми доступными в Beta Basic способами.

Значительно облегчает процесс редактирования программ команда

DELETE [<начало>] **TO** [<конец>]

Она удаляет фрагмент программы со строки с номером <начало> по строку <конец> включительно. В редакторе Spectrum-Бейсик для этого пришлось бы вводить номер каждой подлежащей удалению строки. По умолчанию параметр <начало> принимает значение номера первой строки программы, <конец> — последней строки.

Как и RENUM, команда DELETE «не любит», когда в качестве параметров ей подставляют номера несуществующих строк. В таких ситуациях она выдает сообщение No such line (нет такой строки).

Использование DELETE в качестве оператора в тексте программ позволяет создавать так называемые *оверлейные* программы, то есть программы, состоящие из небольшого блока, постоянно находящегося в памяти компьютера, и фрагментов, подгружаемых по мере необходимости с внешнего носителя. Отработавший фрагмент удаляется оператором DELETE, а на его место (в эти же строки) с помощью MERGE подгружается следующий. Таким образом можно писать программы практически неограниченной длины.

Команда DELETE 0 TO 0 позволяет при необходимости избавиться от нулевой строки, в которой располагаются описания расширенных функций Beta Basic.

Beta Basic не только вводит новые команды, но и совершенствует уже существующие. Одной из таких команд является

LIST [<начало>] **TO** [<конец>]

Она выводит на экран фрагмент программы, ограниченный строками с номерами, заданными значениями параметров <начало> и <конец>.

Команда LIST REF <ключ для поиска> выводит на экран список номеров строк, удовлетворяющих условию <ключа>. Команда LIST DATA распечатает все переменные программы; LIST VAL — только числовые, LIST VAL$ — символьные. Для массивов будут выведены только их размерности.

При помощи команды LIST DEF KEY можно просмотреть список клавиш с приписанными им макросами. Вот такая многоплановая команда!

Завершим рассказ о редакторе Beta Basic описанием очень полезной функции

MEM()

С ее помощью можно определить объем памяти в байтах, доступный для бейсик-программ. Для иллюстрации работы функции выполним программку:

```
10 PRINT MEM(): DIM A(10,100): PRINT MEM()
```

В результате получим два числа, первое из которых на 5008 больше второго. Это вполне естественно, ведь после резервиро-

вания места под массив A() объем памяти, доступный бейсик-программе, значительно уменьшится.

В заключение отметим, что некоторые из описанных в этом разделе ключевых слов при использовании их в качестве операторов в тексте программ могут иметь иной смысл, о чем мы расскажем позже.

ОПЕРАТОРЫ ПРИСВАИВАНИЯ

LET, DEFAULT

В программах, написанных на Beta Basic, допускается одним оператором LET присваивать значения сразу нескольким переменным. Вместо принятой в Spectrum-Бейсике строки

```
LET A=1: LET B=2: LET C=3
```

можно записать:

```
LET A=1, B=2, C=3
```

Список переменных, стоящих следом за одним оператором LET, может иметь произвольную длину. Позже (см. оператор ON) мы покажем, что новая форма записи оператора LET не только ускоряет набор программы, но и имеет серьезное смысловое преимущество перед стандартным форматом.

Beta Basic располагает еще одним оператором присваивания

DEFAULT

В отличие от LET, он задает значение только тем переменным, которые до этого не были определены, то есть упоминаемым в программе впервые. Такое свойство оператора DEFAULT может оказаться полезным в случаях, когда вследствие ветвления программы нельзя предсказать, будет ли переменная определена в данном месте программы или нет:

```
100 LET D=B—A*2
110 IF D=0 THEN LET C=2
120 DEFAULT C=3/D
```

Предполагается, что до этого фрагмента программы переменная C не определена. Оператор DEFAULT помог избежать еще одного условного оператора (120 IF D<>0 THEN LET C=3/D). Другие варианты использования оператора DEFAULT можно найти в разделах, посвященных процедурам и операторам управления программой (см. TRACE, DEF PROC, LOCAL).

Как и LET, оператор DEFAULT допускает одновременную инициализацию нескольких переменных, перечисленных после него через запятую:

```
DEFAULT X=1, Y=2, Z=3
```

ВЫВОД СИМВОЛОВ НА ЭКРАН

OVER, CSIZE, USING, USING$(), KEYWORDS, SCRN$(), PLOT, ALTER

Beta Basic, помимо стандартных, использует дополнительные коды управления выводом на экран:

Окно	Экран	Действие
CHR$ 8	CHR$ 2	курсор влево
CHR$ 9	CHR$ 3	курсор вправо
CHR$ 10	CHR$ 4	курсор вниз
CHR$ 11	CHR$ 5	курсор вверх
—	CHR$ 12	удаление символа (Delete)
—	CHR$ 15	пробел + возврат каретки

В этой табличке коды, указанные в первой колонке, управляют курсором в пределах текущего текстового окна (см. стр. 249), не позволяя курсору «выскочить» за границы окна. Коды из второй колонки игнорируют все ограничения и позволяют курсору «гулять» по всему экрану.

Приведем пример, иллюстрирующий применение этих кодов:

```
10 LET L$=CHR$ 8: LET R$=CHR$ 9
20 LET D$=CHR$ 10: LET U$=CHR$ 11
30 LET a$="1"+D$+L$+"2345"+U$+L$+"6"+U$+L$+"7"+
   L$+L$+L$+"89"
40 PRINT AT 1,0; a$
```

Строка, выведенная на экран, будет иметь вид:

```
 897
1 6
2345
```

Обратим Ваше внимание еще на одно необычное свойство Beta Basic: оператор OVER, кроме привычных 0 и 1, может использоваться с параметром 2. OVER 2 позволяет выводить на экран символы, не затирая и не инвертируя экранного изображения — вывод происходит как бы внакладку.

Beta Basic позволяет устанавливать произвольный размер символов, выводимых на экран. Делается это с помощью оператора

CSIZE <ширина знакоместа>,<высота знакоместа>

После выполнения этого оператора символы при выводе на экран будут вписываться в знакоместа «нового покроя», размеры которых (в пикселях) задаются параметрами <ширина знакоместа> и <высота знакоместа>. Установка шрифта сохраняется до следующего оператора CSIZE. Однако, этот оператор может оказывать и временное действие, если его разместить в операторах PRINT или PLOT подобно INK, PAPER, BRIGHT и FLASH. Проиллюстрировать работу оператора CSIZE может следующая программка:

```
100 CSIZE 50,100
150 PRINT AT 0,0;"Q": REM Огромный символ
200 PAUSE 100: PRINT AT 0,0
```

```
400 FOR i=0 TO 15: REM Каждая новая строка...
600 CSIZE 4+i,8+i: PRINT "QWERTY": REM ...печатается символами
    большего размера
800 NEXT i
900 CSIZE 4,8
```

Рассматривая полученное на экране изображение, можно заметить, что Beta Basic использует только два символьных набора: один — для знакомест шириной до 5 пикселей, другой — для всех остальных. Размер символов может меняться лишь для второго набора, но только дискретно (кратно четырем пикселям). Изменение параметров **CSIZE** между скачками приводит лишь к изменению расстояний между символами и строками (рис. 22).

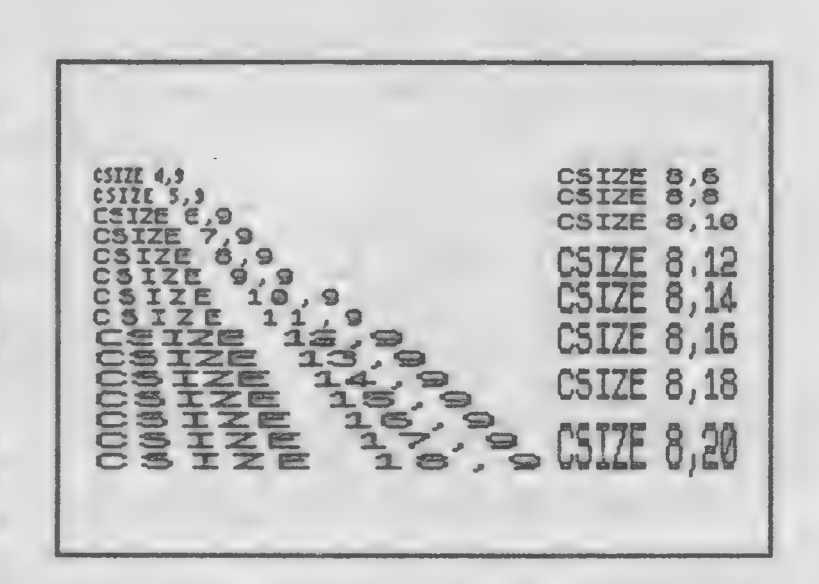


Рис. 22. Шрифты различной ширины и высоты.

При равенстве высоты и ширины символов допустимо указывать лишь один параметр, то есть инструкции CSIZE 8 и CSIZE 8,8 эквивалентны.

После изменения размеров шрифтов операторы позиционирования печати AT и TAB начинают работать в новой сетке знакомест.

Задавая оператором CSIZE сетку из «знакомест-лилипутов» или «знакомест-гулливеров» и при этом активно меняя цвета выводимых символов, можно ненароком получить на экране вермишель из «перенедокрашенных» знаков. Нормальную окраску символов можно установить, лишь задавая параметрам CSIZE значения, кратные восьми. Следует также учитывать, что при установленных размерах знакомест меньше 6 пикселей печатаются только правые половины символов UDG.

Непревзойденной гибкостью в размещении символов на экране обладает оператор

PLOT [список атрибутов;]<x>,<y>[;<строка символов>]

PLOT с новым синтаксисом выводит строку символов, привязывая левый верхний угол первого символа к точке с координатами <x> и <y>. Перед параметрами, задающими координаты точки вывода, можно поставить любые ключевые слова, управляющие атрибутами.

PLOT с опущенной <строкой символов> ничем не отличается от стандартного синклеровского оператора — он ставит на экране точку с координатами <x> и <y>.

Интересной особенностью новой модификации оператора PLOT является то, что при выводе символов в точку с координатами (0, 0) он размещает их ниже 22-й строки экрана, то есть на служебном экране.

Параметр <строка символов> может иметь произвольную длину и включать в себя различные управляющие символы*.

При достижении выводимым текстом правой границы экрана не происходит перехода на следующую строку, вывод продолжается с начала этой же строки. Если длина текста больше ширины экрана, то конец его затрет начало. Для корректного размещения текста необходимо пользоваться управляющими кодами.

Приведем впечатляющий пример применения оператора PLOT:

```
10 FOR N=0 TO 255: PLOT N,N/2;"DEMO": NEXT N
```

Эффект плавно «летящего» по экрану слова возникает в этой программе оттого, что новая позиция вывода строки отличается от предыдущей всего на один пиксель по горизонтали и вертикали. Поскольку изображение большинства символов не плотно вписывается в знакоместо, а имеет окантовку толщиной в один пиксель, то при нахлесте строки, выводимой PLOT-методом со смещением на один пиксель, старый текст затирается новым. К сожалению, этот метод не может использоваться для перемещения символов, изображение которых примыкает к самому краю знакоместа (q, y и некоторых других) — при перемещении этим способом они будут оставлять на экране след.

Спектр применения оператора PLOT весьма широк: это подписи к иллюстрациям, разметка осей графиков, вывод верхних и нижних индексов в математических и химических формулах и т. д. Приведем пример программки, выводящей на экран химическую формулу (рис. 23):

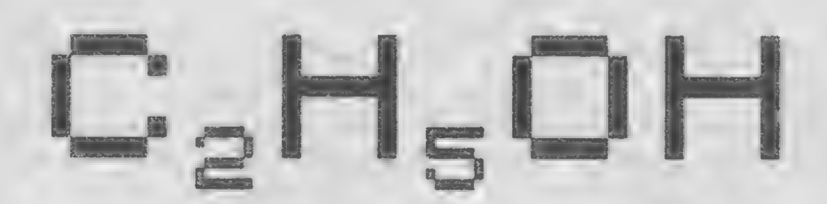


Рис. 23. Пример использования оператора PLOT.

```
10 LET A$="C~2H~5OH"
20 CSIZE 8: LET X=0, Y=170, F=0, A=8
30 FOR N=1 TO LEN A$
40 IF A$(N)="~" THEN LET Y=Y-3, F=1, A=4: CSIZE 4: NEXT N
50 PLOT X,Y;A$(N): LET X=X+A
60 IF F THEN LET Y=Y+3: CSIZE 8: LET F=0, A=8
70 NEXT N: CSIZE 4,8
```

В этом примере управление выводом на экран осуществляется с помощью знака ~. Он указывает на то, что следующий за ним символ необходимо печатать как нижний индекс, то есть после встречи знака ~ в тексте позиция вывода понижается на 3 пикселя и уменьшается размер символа (строка 40).

---

* В операторе PLOT управляющий символ CHR$ 15 перемещает позицию вывода к началу строки.

Beta Basic позволяет преобразовывать числа в символьную форму по заданному шаблону. Осуществить эту операцию можно либо с помощью функции

USING$ (<шаблон>, <числовое выражение>)

– либо вставив ключевое слово USING в строку следом за PRINT:

PRINT USING <шаблон>

<Шаблон> — это символьная строка, состоящая из знаков # и 0, задающих позиции для вывода цифр, и точки (.), определяющей положение десятичной запятой. Например:

```
LET A$=USING$ ("###.#####",12.3456)
```

Если количество знаков в целой части числа меньше количества символов, отведенного для них шаблоном (как в примере), то функция USING$ вернет строку, в которой недостающие слева знаки будут заполнены:

- пробелами, если в шаблоне использованы символы #;
- нулями, если в шаблоне использованы символы 0.



При попытке впихнуть в «прокрустово ложе» шаблона число, у которого целая часть больше, чем позволяет шаблон, оно будет усечено. При этом первым символом в нем будет стоять знак процента (%), сигнализируя о некорректном форматировании.

Недостающие символы в дробной части числа (справа от точки) во всех случаях замещаются нулями. Лишние знаки отсекаются, а число округляется. Приведем примеры различного форматирования числа 123.456*:

Шаблон	Представление
###.###	123.456
######.####	___123.4560
00000.0000	00123.4560
####.#	_123.5
##.##	%2..5
###.#.#	123.4.6
###.#e#	123.4%6
format ###.###	format 123.456
### # format	123.4%%ormat

Из примеров видно, что в маску функции USING$ можно включать и другие символы, но так, чтобы они стояли до шаблона. Только в этом случае гарантируется корректная запись числа. Отметим, что функция USING$ не работает с экспоненциальным представлением чисел.

---

* Для наглядности пробелы в примерах заменены символом подчеркивания (_).

Beta Basic обладает великолепным инструментом преобразования атрибутов экрана. Оператор

ALTER [<список атрибутов 1>] **TO** [<список атрибутов 2>]

ищет на экране знакоместа, атрибуты которых соответствуют списку, приведенному в первом параметре, и заменяет в этих знакоместах атрибуты на новые, представленные списком во втором параметре. ALTER может работать одновременно с любой комбинацией атрибутов:

```
ALTER INK 3, BRIGHT 1 TO PAPER 6, FLASH 1
```

Приведенный оператор, конечно же, не меняет INK на PAPER, а BRIGHT на FLASH. В знакоместах, для которых задано одновременно и INK 3, и BRIGHT 1, он устанавливает PAPER 6 и FLASH 1, не затрагивая других атрибутов.

Левый список атрибутов в операторе ALTER может быть и пустым:

```
ALTER TO PAPER 5, INK 2, BRIGHT 1
```

В этом случае атрибуты переустанавливаются для всего экрана. Но в отличие от операторов INK, PAPER, BRIGHT и FLASH, оператор ALTER задает временные атрибуты. То есть и после выполнения оператора ALTER вывод информации на экран будет продолжаться с ранее установленными постоянными атрибутами.

Пользуясь оператором ALTER, можно весьма эффектно уводить экран в затемнение и выходить из него.

Завершая описание операций вывода на экран символьной информации, рассмотрим функцию

SCRN$ (<строка>,<столбец>)

Она возвращает символ, расположенный на экране в позиции (<строка>,<столбец>), то есть выполняет то же действие, что и функция SCREEN$ Spectrum-Бейсика. Но в отличие от прототипа, SCRN$ распознает на экране не только символы текущего набора, но и символы, определяемые пользователем (UDG), которые выводятся на экран в режиме KEYWORDS 0.

Как SCREEN$, так и SCRN$ работают только в режиме CSIZE 8.

ТЕКСТОВЫЕ ОКНА

WINDOW, CLS

Beta Basic располагает оператором, позволяющим организовать работу с *текстовыми окнами*:

WINDOW <номер окна>[;<x>,<y>,<ширина>,<высота>]

Этот оператор дает возможность открыть до 127 окон*. Параметрами оператора WINDOW задаются: индивидуальный <номер> окна,

* При открытии очередного окна RAMTOP опускается на 15 байтов. В освободившемся пространстве создается область системных переменных, в которой хранятся параметры окна.

координаты левого верхнего угла окна в пикселях — <x> и <y> и его размеры в пикселях — <ширина> и <высота>. Размеры окна округляются до значений, кратных четырем.

Оператор WINDOW с параметрами «по полной выкладке» только задает окна, не производя никаких других действий. Активизировать окно, то есть сделать его текущим, можно, выполнив оператор WINDOW с одним параметром:

WINDOW <номер окна>

После этого операторы PRINT и LIST будет выводить данные только в пределах текущего окна, причем за начало символьных координат — позицию печати (0, 0) — будет принят его левый верхний угол. Это означает, что каждое окно имеет независимую от основного экрана и от других окон нумерацию строк и столбцов.

Для каждого окна задаются свои постоянные атрибуты и свой размер символов. Все параметры окна запоминаются и при возврате к нему (объявлении окна текущим) восстанавливаются. Запоминается также и текущая позиция печати. Вновь открываемые окна по умолчанию имеют следующее состояние: PAPER 0, INK 7, CSIZE 8.

Размеры ранее заданного окна могут быть переопределены. Но смены формата окна не произойдет до тех пор, пока не будет выполнен оператор WINDOW <номер окна> (даже если окно в настоящий момент является текущим). Изменение геометрических параметров окна влечет за собой потерю текущей позиции печати: она возвращается в его левый верхний угол. Цветовое же решение окна при этом сохраняется.

После запуска Beta Basic располагает одним «готовым к употреблению» (заданным и активизированным) окном с номером 0. Оно занимает весь экран, и любая попытка изменить его размеры ни к чему не приведет*. При возврате в нулевое окно из других окон оно не восстанавливает ранее заданный в нем размер символов и постоянные атрибуты экрана и всякий раз с завидным упрямством предлагает стандартный набор: знакоместа 8×8 пикселей** и «окрас» белым по черному.

Можно, не выходя из текущего окна, «почистить» любое другое открытое окно оператором

CLS <номер окна>

Для одновременного закрытия всех окон (кроме нулевого) выполняется оператор

WINDOW ERASE

Проиллюстрируем работу с окнами на примере:

```
10 CSIZE 5,9
20 WINDOW 1;121,175,132,175
```

---

* Нулевое окно отличается еще и тем, что не претендует на участок памяти над RAMTOP.

** Фактически команда WINDOW 0 равносильна установке CSIZE 0, включающей стандартный драйвер быстрого вывода на экран, который игнорирует все оконные установки. Этот режим остался в наследство от предыдущих версий интерпретатора (3.0 и ниже): он устанавливался в них при старте Beta Basic.

```
 30 WINDOW 2;0,175,120,175
 40 WINDOW 1: PAPER 1: CLS: CSIZE 4,8: PRINT AT 0,0;"Window 1"
 50 WINDOW 2: PAPER 2: CLS: PRINT AT 0,0;"Window 2"
 60 WINDOW 1: LIST 10 TO 60
 80 WINDOW 2: CLS 1: LIST
100 WINDOW 0: CLS: LIST
```

В ходе выполнения этой программы на экране одно за другим откроются два окна. В каждое из них поочередно будет выводиться текст символами разных размеров.

ГРАФИЧЕСКИЕ ОПЕРАТОРЫ

DRAW TO, FILL, FILLED(), ROLL, SCROLL

DRAW TO <x>,<y>[,<угловая величина>]

Такой модернизированный синтаксис оператора DRAW позволяет при построении отрезка или дуги использовать в качестве параметров не приращения координат относительно последней выведенной точки (как в Spectrum-Бейсике), а абсолютные координаты конечной точки. Так, инструкции PLOT 10,25: DRAW 30,40 и PLOT 10,25: DRAW TO 40,65 выполнят одинаковые действия.

Смысл третьего параметра — угловой величины в радианах — остается таким же, как и для стандартной команды DRAW.

В Beta Basic все графические операторы приобретают новое полезное свойство: в качестве их параметров можно использовать не только физические координаты экрана (начало отсчета — левый нижний угол основного экрана, единица отсчета — один пиксель), но и математические координаты. То есть графические построения в Beta Basic можно делать в системе координат с произвольной единицей измерения (больше или меньше пикселя), которая определяется специальными переменными **XRG** и **YRG**. Начало отсчета математической системы координат задается другой парой переменных — **XOS** и **YOS** — и может быть расположено в любой точке экрана и даже за его пределами. Так, при XOS=XRG/2 и YOS=YRG/2 начало координат будет находиться в центре экрана, а при XRG<XOS<0 и YRG<YOS<0 — за его пределами. При старте Beta Basic переменные, задающие математическую систему координат, уже определены и имеют стандартные значения: XRG=256, YRG=176, XOS=0 и YOS=0.

Во всех последующих примерах программ, работающих с графикой, предполагается, что переменные XOS, YOS, XRG и YRG имеют стандартные значения.

Beta Basic восполняет существенный недостаток графики Spectrum-Бейсика — вводит операторы заливки замкнутых контуров.

FILL [<список атрибутов>;] <x>,<y>

Оператор FILL осуществляет заливку области экрана, ограниченной замкнутой линией. Задается эта область указанием координат любой точки внутри нее (параметры <x>,<y>). FILL присваивает знакоместам, «задетым» областью заливки, атрибуты,

указанные в <списке атрибутов>. Заливка осуществляется цветом тона либо цветом фона в зависимости от того, какой из атрибутов — INK или PAPER — стоит первым в <списке атрибутов>. Попробуем набрать и выполнить программку:

```
10 CIRCLE 100,100,70
20 FILL INK 5; 100,100
```

и получим на экране голубой круг.

Заливка цветом фона (FILL PAPER) как ластиком стирает ранее нанесенную заливку цветом тона. Дополним программку строкой

```
30 PAUSE 0: FILL PAPER 4; 100,100
```

После нажатия на любую клавишу нарисованный круг перекрасится.

Если опустить у ключевых слов INK или PAPER стоящий после них параметр (код цвета), то закраска будет происходить текущими цветами тона или фона.

Оператор FILL без <списка атрибутов> закрашивает указанную область текущим цветом тона.

После выполнения оператора FILL можно подсчитать количество пикселей, подвергшихся перекраске. Это значение возвращает функция

FILLED()

Она не требует аргументов, результат ее определяется последним выполненным оператором FILL. Функция FILLED() может использоваться, например, для определения площади закрашенной фигуры.

В Beta Basic, как и в других диалектах Бейсика для ZX Spectrum, присутствуют операторы скроллинга *графического окна* экрана*:

ROLL <режим>[,<шаг>][;<x>,<y>;<ширина>,<высота>]
SCROLL [<режим>[,<шаг>][;<x>,<y>;<ширина>,<высота>]]

Оператор ROLL выполняет циклический скроллинг, при котором изображение, пропадающее за границей окна, выплывает с противоположной стороны, а SCROLL — обычный скроллинг, уводящий изображение безвозвратно.

Окно экрана, над которым производятся эти действия, задается четырьмя параметрами: <x> и <y> — координаты в пикселях верхнего левого угла окна; <ширина> — ширина окна, задаваемая **в стандартных знакоместах** (диапазон от 1 до 32); <высота> — высота окна **в пикселях**.

Значения параметров <x> и <y> операторы скроллинга округляют до чисел, кратных восьми, в сторону увеличения.

Если координаты и размеры окна не указывать, то действие оператора будет распространяться на весь экран.

Расстояние в пикселях, на которое будет осуществлено смещение изображения в окне при однократном выполнении операторов

* Окна, в которых осуществляется скроллинг экранного изображения, не следует путать с текстовыми окнами, в которые направляется вывод символьной информации.

скроллинга, задается параметром <шаг>. Допустимый диапазон его значений: от 1 до 256 для горизонтальных перемещений и от 1 до 176 — для вертикальных. По умолчанию параметр <шаг> принимает значение 1.

Параметр <режим> задает направление скроллинга и область его действия: изображение в окне может сдвигаться без смещения атрибутов, со смещением атрибутов, а также возможен сдвиг атрибутов при неизменной картинке. В последнем случае сдвиг производится на 8 пикселей независимо от того, какое значение задано в параметре <шаг>. Зависимость режимов работы операторов ROLL и SCROLL от значения параметра <режим> приведена в табл. 17.

Таблица 17. **Режимы работы операторов ROLL и SCROLL.**

Код режима	Направление	Область действия
1	влево	только атрибуты
2	вниз	только атрибуты
3	вверх	только атрибуты
4	вправо	только атрибуты
5	влево	изображение без атрибутов
6	вниз	изображение без атрибутов
7	вверх	изображение без атрибутов
8	вправо	изображение без атрибутов
9	влево	все изображение
10	вниз	все изображение
11	вверх	все изображение
12	вправо	все изображение

Единственное синтаксическое различие между операторами ROLL и SCROLL заключается в том, что последний можно использовать вообще без параметров. При этом происходит скроллинг всего экрана вверх на 8 пикселей (одну символьную строку) вместе с атрибутами.

УПРАВЛЕНИЕ ПРОГРАММОЙ

IF...THEN...ELSE, GO TO ON, GO SUB ON, ON

В Spectrum-Бейсике ветвление программы реализуется с помощью конструкции IF...THEN и операторов GO TO и GO SUB. В Beta Basic, благодаря дополнению оператора IF...THEN ключевым словом ELSE, появилась возможность в большинстве случаев

вообще отказаться от операторов перехода. Расширенная версия условного оператора записывается так:

IF <условие> **THEN** [:] <блок операторов 1> [:]
[**ELSE** [:] <блок операторов 2>]

В зависимости от того, истинно или ложно <условие>, выполнится, соответственно, либо <блок операторов 1>, либо альтернативный <блок операторов 2>.

Вся конструкция должна быть записана в одну строку.

Каждый из блоков операторов внутри себя может содержать еще конструкции IF...THEN...ELSE, причем глубина таких вложений неограничена. В этом случае, если на каком-либо уровне вложения <блок операторов 1> не предполагает наличия альтернативного <блока операторов 2>, то на этот уровень нужно принудительно ввести «пустой» ELSE. Это необходимо сделать во избежание неоднозначности в определении, к какому именно из IF...THEN относится данный ELSE:

```
100 IF L>15 THEN:
    IF J<L THEN LET K=L*J:
    ELSE:
    ELSE LET K=0
```

Кроме конструкции IF...THEN...ELSE, Beta Basic располагает и другими операторами, организующими разного рода ветвления программ.

GO TO ON <числовая переменная>;<список адресов перехода>

Параметр <список адресов перехода> представляет собой перечень номеров строк, записанных через запятую, например:

```
10 GO TO ON J;70,150,345
```

Значение первого параметра задает порядковый номер адреса, по которому нужно передать управление. Таким образом, приведенная программная строка равносильна фрагменту:

```
10 IF J=1 THEN GO TO 70
20 IF J=2 THEN GO TO 150
30 IF J=3 THEN GO TO 345
```

<Список адресов перехода> может иметь произвольную длину. Если значение <числовой переменной> превысит количество элементов в списке, оператор будет проигнорирован, управление передастся следующей за ним строке. При отрицательном значении <числовой переменной> знак «минус» отбрасывается.

GO SUB ON <числовая переменная>;<список адресов перехода>

Этот оператор отличается от предыдущего тем же, чем обычный GO SUB отличается от GO TO.

Обе инструкции дают возможность компактно организовывать ветвление программы и просто незаменимы при создании разного рода меню.

Локальные переходы в пределах одной строки позволяет осуществлять оператор варианта

ON <числовая переменная>:<оператор>:[<оператор>: ...]

Из всего списка операторов, расположенных за ON, будет выполнен только один — тот, чей порядковый номер в строке задан значением параметра <числовая переменная>. После этого управление будет передано следующей за конструкцией ON строке программы (разумеется, если не произошло принудительного перехода или вызова подпрограммы*).

Основное достоинство оператора ON — его универсальность. К примеру, с его помощью можно организовать многовариантное переопределение переменных (используя способность оператора LET обслуживать список переменных произвольной длины):

```
100 INPUT J
110 ON J: LET A=484, B=50, C=88: LET A=242, B=12, C=82:
    LET A=212, B=85, C=06
120 PRINT A, B, C
```

Интересным свойством будет обладать приведенная в примере конструкция, если ее поместить в цикл FOR...NEXT, в котором переменная цикла и параметр оператора ON совпадают. Одним словом, спектр применения оператора ON ограничен лишь фантазией программиста.

ОБРАБОТКА ОШИБОК, ТРАССИРОВКА, ЧАСЫ

ON ERROR, TRACE, CLOCK, TIME$ ()

ON ERROR <номер строки>

Этот оператор не производит явных действий в момент выполнения. Он лишь указывает интерпретатору номер строки, с которой начинается подпрограмма обработки ошибок. Если во время выполнения программы возникнет ситуация, приводящая к выдаче сообщения об ошибке, управление будет немедленно передано подпрограмме, начинающейся с указанной строки. Операторы ON ERROR 0, RUN и CLEAR восстанавливают обычный режим обработки ошибок.

Задать подпрограмму обработки ошибок можно и непосредственно сразу после оператора ON ERROR, используя следующий формат:

ON ERROR: <оператор>:<оператор>:...<оператор>: RETURN

Для наглядности вместо набора операторов можно просто указать имя процедуры обработки ошибок, а саму процедуру оформить отдельным программным модулем.

---

* Стоящий следом за ON список операторов может содержать также обращения к процедурам, вызываемым по именам (их мы рассмотрим несколько позже). В этом случае он будет выступать уже в следующей «весовой категории» операторов управления программой после GO SUB ON.

Проанализировать причины, приведшие к ошибке, и определить место в программе, где она произошла, помогут специальные системные переменные Beta Basic. К ним, в отличие от обычных системных переменных ZX Spectrum, имеется доступ по именам, как к обычным числовым переменным: переменная с именем ERROR содержит код возникшей ошибки (информация о кодах ошибок приводится в Приложении 3). Переменные LINO и STAT указывают, соответственно, номер строки и номер оператора в строке, где была прервана работа программы. Эта информация позволяет предусмотреть подходящую реакцию на ошибку в зависимости от вызвавшей ее причины:

```
  10 ON ERROR 1000: LET J=1
  30 FOR K=1 TO 10000
  40 PLOT INT (257*RND),INT (178*RND)
  50 NEXT K: STOP
1000 REM Начало подпрограммы обработки ошибок
1010 BEEP 1,69
1020 IF ERROR=11 AND LINO=40 THEN PRINT AT 0,0;J: J=J+1:
     ELSE BEEP 0.5,12
1030 RETURN
```

Фрагмент программы (строки 30...50) выводит на экран точки со случайными значениями координат, которые могут выходить за допустимые пределы (X>255, Y>175). Обычно попытка поставить точку за пределами экрана приводит к ошибке с выдачей сообщения Integer out of range. Но в данном примере строка 10 предписывает в случае ошибки обращаться к подпрограмме, начинающейся со строки 1000. В ней анализируется, вызвано ли прерывание ошибкой выхода за пределы экрана, и если «да», то на экран выводится номер J такого «выхода», после чего управление возвращается в основную программу. Если попытаться сгенерировать ошибку с другим кодом, например, нажать **Break**, то программа, подав звуковой сигнал (оператор BEEP в строке 1020), продолжит свою работу.

На время выполнения подпрограммы обработки ошибок «антиошибочный иммунитет» отключается. Поэтому, если не предусмотреть необходимой защиты, то при нажатии клавиши **Break** программа остановится, поскольку переход к подпрограмме обработки ошибок произойдет еще до того, как будет отпущена клавиша. Во избежание такой ситуации в начале подпрограммы обработки ошибок нужно всегда организовывать задержку, достаточную для того, чтобы успеть освободить клавишу **Break**. В нашем примере это сделано в строке 1010 с помощью оператора BEEP, звучащего в неслышимом диапазоне частот (BEEP не реагирует на нажатие **Break**).

Вернуться из подпрограммы обработки ошибок можно двумя способами. Первый способ — по RETURN (как в приведенном примере), когда управление передается оператору, следующему за тем, который вызвал ошибку*, а ON ERROR опять активен — готов к

---

* В нашем примере это оператор NEXT K.

охоте за ошибками. Второй способ — по CONTINUE, после которого действие оператора ON ERROR отключается, а управление (за редкими исключениями) вновь возвращается оператору, вызвавшему ошибку. Повторное выполнение этого оператора приведет к привычному останову программы с выдачей сообщения об ошибке. Пользуясь этими двумя способами, в подпрограмме обработки ошибок можно предусмотреть выбор: в зависимости от места возникновения ошибки или ее типа работа программы продолжается либо прекращается.

Следует помнить, что, возвращаясь из подпрограммы по CONTINUE, необходимо предварительно снять со стека GO SUB адрес возврата из подпрограммы, так как оператор CONTINUE сам этого не делает. Операцию по очистке вершины стека GO SUB производит оператор POP (его работа описана далее). Поэтому возврат по CONTINUE в приведенном выше примере должен выглядеть так:

```
1030 POP: CONTINUE
```

После того как в программе встретится оператор трассировки

TRACE <номер строки>

или

TRACE: <оператор>:<оператор>:...<оператор>: **RETURN**

— интерпретатор, после каждого выполненного оператора, будет делать переход к подпрограмме, расположенной с указанного <номера строки> или размещенной непосредственно за оператором TRACE после двоеточия. Это будет происходить до тех пор, пока в программе не встретится оператор TRACE 0 или CLEAR. Пользуясь этим свойством, можно замедлять выполнение программы, отслеживать по ее ходу изменения значений переменных, следить за состоянием клавиатуры (не нажималась ли какая-либо клавиша) и т. д. Поместим в начало предыдущего примера строку

```
1 TRACE 2000
```

— а также добавим следующий фрагмент:

```
2000 REM Подпрограмма трассировки
2005 WINDOW 1;140,175,115,64
2010 WINDOW 1: CLS: CSIZE 4,8: DEFAULT K=0
2020 LIST LINO TO LINO: PRINT AT 7,0;"K=";K
2030 PRINT #0; AT 0,0; "String ";LINO;":";STAT;
2040 PAUSE 0: CLS: WINDOW 0: CSIZE 4,8
2050 RETURN
```

Теперь можно выполнить программу в пошаговом режиме. В правом верхнем углу экрана (строки 2005, 2010) будут выводиться исполняемая в настоящий момент строка программы и текущее значение переменной K (строка 2020). В нижней части экрана будет выводиться номер строки и номер выполняемого оператора в ней (строка 2030). Для осуществления следующего шага необходимо нажимать клавишу **Enter** (строка 2040). Заменив в 2040-й строке оператор PAUSE 0 на IF INKEY$="1" THEN POP: STOP, можно сделать так, чтобы программа работала без пауз и останавливалась при нажатии клавиши **1**.

Если нужно в ходе выполнения программы наблюдать за изменениями значений переменных, целесообразно в начале подпрограммы трассировки инициализировать их оператором DEFAULT. Иначе возможен останов программы по ошибке с сообщением Variable not found.

CLOCK <число|строка символов>

Этот оператор управляет часами с «боем», встроенными в Beta Basic: устанавливает текущее время, высвечивает его в верхнем правом углу экрана, задает контрольный час, минуту и секунду, когда необходимо подать звуковой сигнал и/или передать управление подпрограмме.

При загрузке Beta Basic устанавливается время 00:00:00. Для переустановки времени нужно выполнить оператор CLOCK в следующем формате:

CLOCK <"HH[:]MM[:]SS">

– где HH — часы, MM — минуты, SS — секунды. Разделители (двоеточия) между цифрами можно опускать, достаточно ввести 6 цифр. В случае, если количество цифр окажется меньшим, вместо недостающих будут подставлены нули. Так, например, CLOCK "173" установит время 17:30:00.

Для настройки будильника, который в назначенное время подаст звуковой сигнал и/или передаст управление подпрограмме, к строке символов, задающей время, в первую позицию добавляется символ A:

```
CLOCK "A09:45"
```

Режим работы часов задается оператором

CLOCK <код режима>

Всего имеется 8 режимов работы, они перечислены в табл. 18.

Таблица 18. **Режимы работы оператора CLOCK.**

Код режима	Индикация времени	Подача звукового сигнала	Переход к подпрограмме
0	–	–	–
1	+	–	–
2	–	+	–
3	+	+	–
4	–	–	+
5	+	–	+
6	–	+	+
7	+	+	+

Так, к примеру, CLOCK 7 установит режим, в котором текущее время будет высвечиваться на экране, а в заданный момент прозвенит будильник и произойдет обращение к подпрограмме. Номер строки, с которой начинается подпрограмма, задается оператором вида:

CLOCK <номер строки>

— где <номер строки> — число от 8 до 9999 (числа от 0 до 7 зарезервированы под номера режимов). Возможно также размещение подпрограммы непосредственно за оператором CLOCK через двоеточие:

CLOCK: <оператор>:<оператор>:...<оператор>: RETURN

Следует учитывать, что часы, прежде чем «зазвенеть» при достижении контрольного времени, ждут окончания выполнения текущей строки программы, что иногда может длиться долго (например, ввод с клавиатуры, выполнение оператора PAUSE).

Часы Beta Basic поддерживают также символьную функцию



TIME$ ()

— возвращающую значение текущего времени в виде строки символов, в которой часы, минуты и секунды разделены знаком двоеточия.

Нужно иметь в виду, что часы приостанавливают отсчет времени при генерации звукового сигнала оператором BEEP и операциях с внешними устройствами (в частности, магнитофоном).

Описанные свойства оператора CLOCK позволяют создавать программы, которые через заданные интервалы времени приостанавливают работу, проделывают некие запрограммированные действия, а затем снова продолжают свое дело. Предположим, что к компьютеру подключен цифровой прибор, с которого необходимо снимать данные каждые 10 секунд. Основная программа производит обработку данных (к примеру, выводит на экран диаграмму показаний прибора, делает статистический анализ и т. д.). А подпрограмма, к которой происходит обращение в заданное время, считывает очередное показание прибора (например, из порта 104) и помещает его в специальный массив. Последний служит кольцевым буфером для согласования времени обработки данных в основной программе с частотой их поступления из подпрограммы. Подпрограмма также устанавливает новое значение контрольного времени для снятия очередного показания прибора. Программа обслуживания прибора может выглядеть так:

```
 10 CLOCK 6: REM Установка режима работы часов
 20 CLOCK:PortRead: RETURN: REM задание имени процедуры
 30 LET Max=3000: REM Задание размера массива-буфера
 40 DIM A(Max): REM Инициализация массива
 50 LET MIdx=0, SIdx=0: REM Инициализация указателей массива
 60 PROC PortRead: REM Вызов процедуры считывания
100 LET MIdx=MIdx+1: IF MIdx>Max THEN LET MIdx=1
```

```
 120 IF A(MIdx)=0 THEN GO TO 120: REM Ожидание новых данных
 130 LET Value=A(MIdx): LET A(MIdx)=0

  ... (обработка показаний прибора)

7000 GO TO 100: REM Возврат к началу цикла

9000 DEF PROC PortRead: REM Процедура считывания показаний
     прибора
9010 LET SIdx=SIdx+1: IF SIdx>Max THEN LET SIdx=1
9030 LET A(SIdx)=IN 104: REM Считывание данных из порта
9040 CLOCK "00:00:00": REM Сброс часов
9060 CLOCK "A00:00:10": REM Время следующего «звонка»
9070 END PROC
```

В данном примере первые показания снимаются при принудительном обращении к подпрограмме считывания данных (строка 60), а все остальные — при вызове этой подпрограммы по прерыванию от таймера. Если основная программа опережает поступление данных, она ожидает их в бесконечном цикле (строка 120).

Другой пример использования оператора CLOCK и функции TIME$ можно найти в разделе, описывающем циклы типа DO...LOOP (стр. 266).

ПРОЦЕДУРЫ

DEF PROC, LOCAL, REF, DATA, READ, READ LINE, ITEM(), END PROC, PROC, LIST PROC

В Beta Basic по сравнению с другими диалектами Бейсика для ZX Spectrum инструмент процедур реализован, пожалуй, наиболее удачно (см. стр. 176, 222).

Процедуры описываются блоком текста программы, состоящим из оператора-заголовка, собственно тела и оператора завершения процедуры:

DEF PROC <имя процедуры>
        [<список формальных параметров>|**DATA**]
DEFAULT <задание значений формальных параметров по умолчанию>
[**LOCAL** <список локальных переменных и массивов номер 1>]
        <операторы, [**READ** [**LINE**] <переменная>]>
[**RETURN**]
...
[**LOCAL** <список локальных переменных и массивов номер N>]
        <операторы>
[**RETURN**]
...
END PROC

<Имя процедуры> может иметь произвольную длину и состоять из любых символов (кроме псевдографики, UDG, символов пробела и ключевых слов), но начинаться оно должно обязательно с буквы.

При этом оно может совпадать с именами переменных как в основной программе, так и в самой процедуре. Блок описания процедуры может находиться в любом месте программы, но оператор заголовка обязательно должен быть первым оператором в строке.

Возврат из процедуры может происходить в любой ее точке по оператору RETURN либо в конце — по END PROC. Оператор END PROC, помимо функции возврата, выполняет еще и роль маркера конца тела процедуры (этим он и отличается от RETURN): если процедура встроена в середину текста основной программы, последняя, наткнувшись на заголовок процедуры, «перепрыгнет» через нее. Точкой «приземления» послужит оператор, следующий за END PROC. Если же интерпретатор не сможет найти конца процедуры, он прервет выполнение программы сообщением No END PROC.

Листинг процедуры может быть выведен на экран отдельно от всей программы оператором

LIST PROC <имя процедуры>

Вызов процедуры осуществляется следующим образом:

[**PROC**] <имя процедуры> [<список фактических параметров>]

В этих конструкциях полностью отсутствует какая бы то ни была привязка к номерам строк. Эта особенность процедур в сочетании с возможностью принимать данные из основной программы посредством списка параметров позволяет создавать автономные программные модули. Однажды разработанные, они могут быть использованы в других программах практически без изменений. Достигается это объявлением внутренних переменных процедуры *локальными*, что позволяет уйти от проблемы совпадения их имен в процедуре и основной программе. Причем в Beta Basic (в отличие, например, от Laser Basic и MegaBasic) они действительно «до конца» локальны.

Переменные объявляются локальными с помощью оператора LOCAL, который может располагаться в любом месте процедуры и повторяться с разным списком переменных не один раз. Список представляет из себя перечень имен переменных, разделенных запятыми. Переменная, указанная в LOCAL, теряет свое прежнее значение (если она его до этого имела) и становится неопределенной. Ее можно заново инициализировать и использовать в теле процедуры с новым содержанием. После возврата в основную программу ей будет возвращено прежнее значение:

```
 10 LET A=1
 20 PROC Test
 30 PRINT A,D: STOP
100 DEF PROC Test
110 LET D=5: LOCAL A: DEFAULT A=10
120 PRINT A,D
130 END PROC
```

В процедуре Test вводится новая переменная D и объявляется локальной определенная в основной программе переменная A (строка 110). Оператор PRINT (строка 120) напечатает два числа: 10

и 5, что соответствует значениям переменных A и D в процедуре. После возвращения из процедуры оператор PRINT (строка 30) огласит новую обстановку: переменная A без потерь пережила «прогулку» в процедуру и сохранила прежнее значение; переменная D вынесла значение из процедуры, так как не была объявлена в ней локальной.

Кроме переменных, в процедуре можно объявлять локальными и массивы (еще один плюс Beta Basic), после чего их необходимо инициализировать оператором DIM. Процедура, в которой встретятся такие строки:

```
110 LOCAL A(),S$
120 DIM A(10,20): DIM S$(5,40)
```

— создаст два локальных массива: числовой с именем A и символьный с именем S$. Если последний не объявлять как массив, процедура будет воспринимать S$ как простую символьную переменную. Значения, присвоенные в процедуре элементам локальных массивов, при возврате из нее будут утрачены, а содержимое одноименных массивов в основной программе восстановлено.

В случае многократного вложения процедур (когда одна вызывает другую, та, в свою очередь, третью и т. д.) объявленные на одном из уровней вложения локальные переменные и массивы могут использоваться всеми процедурами более низкого уровня, оставаясь «невидимыми» для «верхних» процедур.

Как мы уже упоминали, список формальных параметров процедуры указывается после ее имени в операторе-заголовке. Значения *формальным* параметрам присваиваются посредством приведения в операторе вызова процедуры списка *фактических* параметров, размещенных через запятую после имени процедуры. Фактические параметры могут быть заданы как в явном виде, то есть числовыми или символьными константами, так и переменными, элементами массивов, выражениями. Разумеется, все переменные и элементы массивов должны быть определены к моменту вызова процедуры.

При вызове процедуры значения фактических параметров присваиваются по порядку соответствующим переменным из списка формальных параметров. Все переменные, входящие в список формальных параметров, автоматически становятся локальными.

В общем случае допустимо, чтобы число фактических параметров в операторе вызова процедуры было меньше числа формальных параметров из описания процедуры. При этом «лишние» переменные из заголовка процедуры просто останутся в ней неопределенными. Однако имеется возможность с помощью описанного ранее оператора DEFAULT задать значения пропущенных параметров, которые будут приниматься ими по умолчанию. Если же Вы хотите при вызове процедуры опустить один или несколько фактических параметров, стоящих в середине списка, то место в списке каждого из них необходимо «пометить» лишней запятой.

В том случае, когда вообще невозможно заранее предсказать количество передаваемых в процедуру фактических параметров, список формальных параметров в ее заголовке заменяется одним ключевым словом DATA. Считывание в процедуре значений из такого «резинового» списка необходимо осуществлять оператором

READ. Для того чтобы определить тип очередного элемента данных и обнаружить конец списка, используется функция ITEM(). Она не имеет параметров и возвращает 1, если в списке параметров следующим стоит символьное значение, 2 — если числовое значение, и «разрешается» нулем при достижении конца списка. Избежать использования кавычек при пересылке строковых значений позволяет ключевое слово LINE, помещаемое в процедуре после READ (например, READ LINE A$)*.

Приведем пример пакета процедур, использующих описанные выше формы приема параметров:

```
 80 DEF PROC Prima DATA: LOCAL A,B,C
 90 DO WHILE ITEM()<>0
100 READ A: READ B: READ C
110 LET D=B*B-4*A*C
120 IF D>=0 THEN PROC Real: ELSE PROC Complex
130 LOOP: END PROC
140 DEF PROC Real: LOCAL X1,X2
150 LET X1=(-B+SQR D)/(2*A)
160 LET X2=(-B-SQR D)/(2*A)
170 PROC Root ,X1,X2: REM Первый параметр опущен
180 END PROC
190 DEF PROC Complex: LOCAL Re,Im
200 LET Re=-B/(2*A), Im=SQR ABS D/(2*A)
210 PROC Root 1,Re,Im
220 END PROC
230 DEF PROC Root Cmpl,A,B
240 DEFAULT Cmpl=0: PRINT "Roots:",
250 IF Cmpl THEN PRINT "X1=";A;"+";B;"i","X2=";A;"-";B;"i":
    ELSE PRINT "X1=";A,"X2=";B
260 END PROC
```

Этот пакет процедур вычисляет корни X1 и X2 квадратного уравнения A*X↑2+B*X+C. Процедура Prima в цикле (строка 90) считывает очередные коэффициенты уравнения (строка 100), рассчитывает по ним дискриминант (строка 110) и, в зависимости от его знака, вызывает одну из двух процедур: Real или Complex (строка 120). Каждая их них в свою очередь обращается к процедуре вывода результатов Root. Но значения, передаваемые в эту процедуру, имеют разный смысл: в первом случае это действительные корни уравнения, во втором — действительная и мнимая части сопряженных комплексных корней. Поэтому в процедуру Root передается признак «комплексности» корней Re: в зависимости от его значения, процедура выбирает вариант вывода на экран (строка 250). Обращение к этому пакету может выглядеть, например, так: PROC Prima 3,10,1,1,2,3 Количество параметров при этом должно быть кратно трем.

* Конструкция READ LINE A$ и функция ITEM() работают и с обычными DATA-списками. К сожалению, при этом ITEM() не в состоянии определить тип *первого* элемента очередного DATA-списка. Следовательно, и после выполнения оператора RESTORE она тоже вернет нулевое значение. Впрочем, этот недостаток в работе функции несложно обойти. Достаточно поместить в самое начало каждого DATA-списка элемент одного и того же типа.

Beta Basic позволяет не только пересылать значения параметров из вызывающей программы в процедуру, но и принимать из нее результаты расчетов, не прибегая при этом к глобальным переменным. Это достигается путем передачи процедуре *ссылки* на соответствующий фактический параметр. Если при передаче параметров обычным порядком (по значению) в процедуре создается временная область локальных переменных, в которую и копируются значения фактических параметров, и которая уничтожается при возврате из процедуры, то при передаче значения по ссылке процедура «получает в руки» лишь указатель на соответствующий фактический параметр (его адрес в основной области переменных). Работая с таким локальным параметром-ссылкой, процедура фактически напрямую «общается» с соответствующей переменой из основной программы, лишь «называя» ее своим, локальным именем. Все изменения, производимые с такой локальной переменной, автоматически отражаются на содержимом ее прообраза из вызывающей программы. Для указания того, что передача параметра производится по ссылке, используется ключевое слово REF (reference), которое помещается перед каждым соответствующим формальным параметром в заголовке процедуры:

```
10 LET A=5, B=10, X=1, Y=2: PRINT A,B,X,Y
20 PROC Test A,B: PRINT A,B,X,Y
40 DEF PROC Test REF X, REF Y: LOCAL Z
50 LET Z=X, X=X+Y, Y=Z*Y
60 END PROC
```

В этом примере все действия, производимые в процедуре над локальными переменными X и Y, приводят к изменению соответствующих им переменным A и B: значения последних, выводимые операторами PRINT до вызова процедуры и после нее, будут различными, в то время как переменные X и Y в основной программе не изменятся, несмотря на то, что одноименные им переменные были использованы в процедуре.

Такой способ пересылки параметров позволяет передавать процедуре массивы, указывая после REF их имена с пустыми скобками. Налицо серьезная экономия памяти: процедура не создает копии массива, а работает с оригиналом, лишь пользуясь его локальным «псевдонимом».

При обращении к процедуре можно опускать служебное слово PROC, таким образом фактически расширяя арсенал операторов Beta Basic. К примеру, можно написать собственную процедуру с именем BOX, рисующую на экране прямоугольник, и обращение к ней по синтаксису ничем не будет отличаться от аналогичных операторов, скажем, CIRCLE. Заметим однако, что эта особенность интерпретатора может сыграть с программистом злую шутку: поскольку имя процедуры может состоять практически из любого набора символов, редактор Beta Basic способен «проглотить» какую-нибудь синтаксически неверную строку, приняв ее за экзотическое имя процедуры.

Beta Basic допускает *рекурсию* (обращение процедуры к самой себе) практически неограниченное количество раз. Но на

практике, вследствие переполнения стека адресов возврата и «размножения» локальных областей переменных, количество рекурсий едва ли сможет превысить несколько сотен (при этом скорость работы программы значительно снижается). Этого эффекта можно избежать, если все параметры в рекурсивную процедуру передавать по ссылке.

ЦИКЛ DO...LOOP

DO, LOOP, UNTIL, WHILE, EXIT IF

Beta Basic в дополнение к имеющемуся в базовой версии циклу типа FOR...NEXT располагает еще одной подобной конструкцией — циклом DO...LOOP. Как и его «собрат», он также состоит из оператора начала цикла, тела и оператора завершения цикла. Кроме того, в теле цикла могут располагаться один или несколько операторов выхода из цикла по условию. В общем виде конструкция цикла DO...LOOP выглядит так:

DO [**WHILE** <лог. выражение> | [**UNTIL** <лог. выражение>]

...

[**EXIT IF** <лог. выражение>]

...

LOOP [**WHILE** <лог. выражение>] | [**UNTIL** <лог. выражение>]

Специфика цикла DO...LOOP заключается в том, что выход из него происходит не после определенного количества повторений тела цикла, как в FOR...NEXT, а по выполнении одного из условий, заданных логическими выражениями в начале цикла после оператора DO, в конце цикла после оператора LOOP или одним из операторов EXIT IF в теле цикла. Во всех случаях после завершения цикла управление передается оператору, стоящему после LOOP.

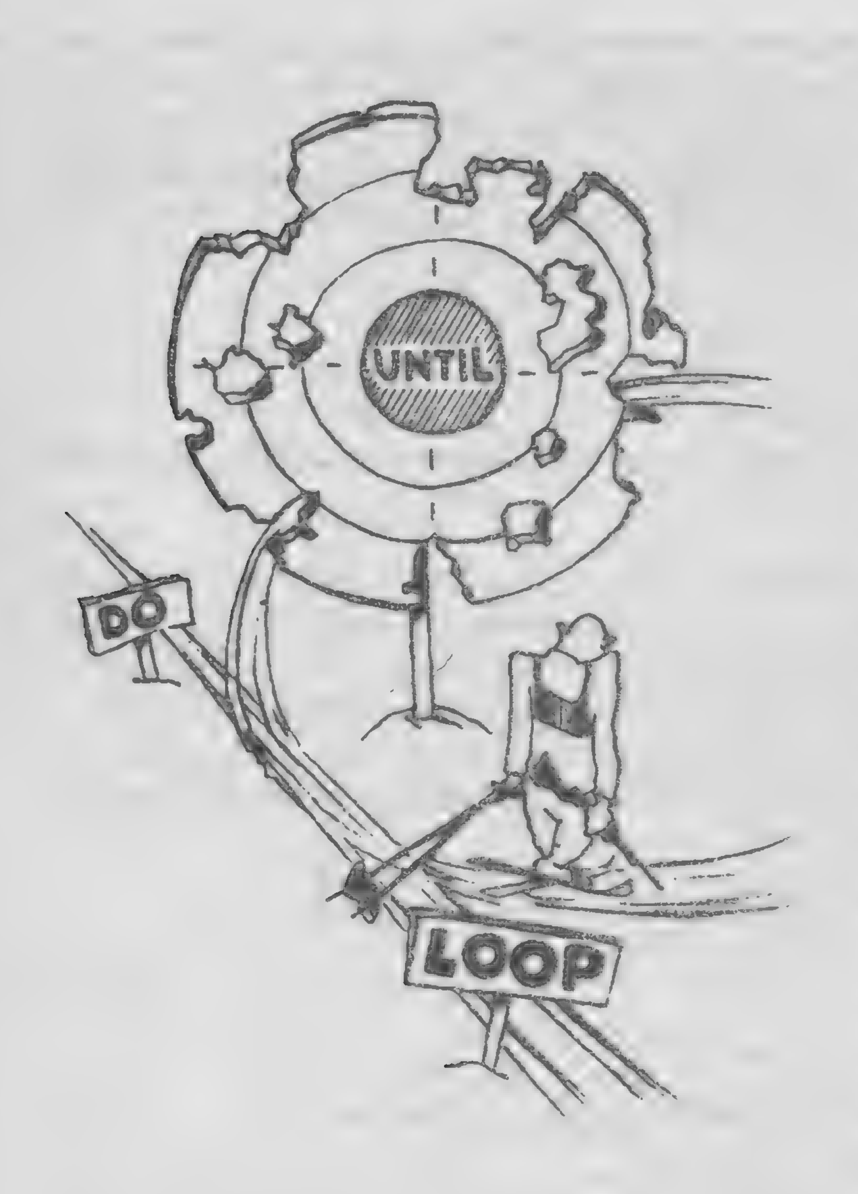
Beta Basic предусматривает два типа критериев выхода из цикла, похожих друг на друга «с точностью до наоборот»:

WHILE <лог. выражение> предписывает циклу повторяться, пока <лог. выражение> истинно. Таким образом, WHILE задает критерий продолжения цикла;

UNTIL <лог. выражение> допускает повторение цикла до тех пор, пока <лог. выражение> не станет истинным, то есть предписывает прекратить цикл, как только будет соблюдено условие. Тем самым UNTIL задает критерий окончания цикла.

Налицо очевидная избыточность, поскольку UNTIL <лог. выражение> и WHILE NOT <лог. выражение> — суть одно и то же. Их одновременное присутствие в языке связано с желанием добиться большей наглядности при написании программ.

Место расположения в цикле инструкций WHILE и UNTIL определяет, где будет проверяться условие выхода из него. В случае, если цикл делает заведомо больше одного прохода, абсолютно безразлично, где именно будет размещен критерий его продолжения или окончания. Если же заранее неизвестно, какое значение — «истина» или «ложь» — будет иметь <лог. выражение> в момент первого



вхождения в цикл, то важно, где это будет проверяться — в заголовке цикла (*предусловие*) или же в его конце (*постусловие*). Во втором случае, независимо от значения <лог. выражения>, цикл сделает, по крайней мере, один проход, что может оказаться нежелательным. Поэтому рекомендуем использовать циклы DO...LOOP преимущественно с предусловием — это застрахует от возможных ошибок. Тем не менее, в ситуации, когда анализируемый в <лог. выражении> параметр вводится впервые в теле цикла, необходимо завершение цикла по условию, заданному оператором LOOP. Именно такой вариант мы и предлагаем в приводимом ниже примере. В нем осуществляется поиск одного из корней квадратного уравнения методом Ньютона (метод касательных):

```
10 INPUT "Коэфф. уравнения: A=";A,"B=";B,"C=";C
20 INPUT "Начальное приближение X="; X
30 LET Eps=0.001: REM Погрешность
40 CLOCK "00:00:00": REM Установка времени
50 DO
60 IF A*X+B=0 THEN POP: GO TO 140: REM Выход из цикла, если
   деление на ноль
70 LET F=(A*X*X+B*X+C)/(A*X+B)
80 LET Absent=(VAL (TIME$ ()(7 TO 8)) > 30)
90 EXIT IF Absent: REM Выход, если зацикливание
100 LET X=X-F
110 LOOP UNTIL ABS (F) < Eps
120 IF Absent THEN PRINT "Уравнение корней не имеет":
    ELSE PRINT "Корень уравнения X=";X
130 GO TO 10
140 PRINT "Попытка деления на ноль!"
150 GO TO 10
```

Запросив вначале все необходимые параметры, программа задает погрешность вычислений и обнуляет часы. После этого в цикле (строки 50...110) производятся вычисления, а в конце цикла проверяется, достигнута ли заданная точность. Как только погрешность расчета станет меньше установленной величины Eps (строка

110), цикл завершит свою работу. Если уравнение не имеет решений, программа зациклится. На этот случай предусмотрен контроль времени (строка 80), и если программа работает больше полминуты (время заведомо достаточное, чтобы найти корень), происходит принудительный выход по EXIT IF (строка 90).

При входе в цикл интерпретатор помещает в стек GO SUB адрес начала цикла. По достижении оператора LOOP это значение снимается со стека, и по нему вновь отыскивается соответствующий данному LOOP заголовок DO. При выходе из цикла по EXIT IF адрес возврата автоматически снимается со стека. Выход же из тела цикла посредством GO TO или GO TO ON необходимо предварить принудительной чисткой вершины стека оператором POP (см. далее). В нашей программе есть пример и такого перехода (строка 60): он предотвращает аварийный останов программы при попытке деления на ноль.

Во избежание конфликтов со стеком запрещены переходы извне в тело цикла, минуя его заголовок. Если пренебречь этим замечанием, выполнение программы будет прервано с выдачей сообщения LOOP without DO или No POP data. Подобно циклам FOR...NEXT циклы DO...LOOP допускают многократное вложение одного в другой.

Ранее неоднократно упоминался оператор POP. Рассмотрим его подробнее. В общем виде он записывается так:

POP [<числовая переменная>]

Этот оператор снимает с вершины стека GO SUB значение номера строки возврата и присваивает его <числовой переменной>, если она указана. Оператор POP позволяет корректно выйти из подпрограммы, процедуры PROC или цикла DO...LOOP без дезориентации стека. Пользуясь этим оператором, можно возвратиться из процедуры, вызванной, в свою очередь, из другой процедуры, сразу к месту первого вызова.

Снятый со стека и присвоенный <числовой переменной> адрес возврата можно в дальнейшем использовать для перехода в операторах GO TO и GO TO ON, чтобы попасть в то место, откуда была вызвана процедура.

РАБОТА С ПАМЯТЬЮ

DPOKE, DPEEK(), POKE, MEMORY$()

Beta Basic, как и любой другой серьезный диалект Бейсика, может оперировать не только одно-, но и двухбайтовыми числами.

Для записи в память двухбайтового числа достаточно выполнить оператор

DPOKE <адрес>,<число>

– а для чтения двухбайтового числа применить функцию

DPEEK (<адрес>)

Более того, Beta Basic располагает также средствами записи в память целых строк символов (точнее, кодов этих символов), начиная с указанного адреса. Для этих целей используется все тот же POKE, но с новым синтаксисом:

POKE <адрес>,<строка символов>

Предусмотрена и обратная функция, воспринимающая всю память как одну большую строку кодов символов и выделяющая участок памяти, который можно, например, скопировать в символьную переменную:

MEMORY$ () (<адрес начала> **TO** <адрес конца>)

Все эти новые средства позволяют быстро обрабатывать большие порции памяти. Впечатляющим примером может послужить такая программка:

```
10 LET A$=STRING$ (6144,CHR$ BIN 01010101)
20 LET B$=STRING$ (6144,CHR$ BIN 10101010)
30 DO: POKE 16384,A$: POKE 16384,B$: LOOP
```

В ней функция STRING$ (к подробному рассмотрению ее мы вернемся позже) формирует строки длиной по 6144 символа, коды которых представляют собой в двоичном виде чередующиеся нули и единицы. А оператор POKE записывает эти коды в экранную область памяти (6144 байта — размер этой области без атрибутов). Результатом работы такой программки будут быстро бегущие по экрану вертикальные полоски.

Пользуясь оператором POKE и функцией MEMORY$, можно запоминать экранные картинки как символьные строки, а затем снова выводить их на экран:

```
10 PLOT 60,60: DRAW 100,100,4000
20 LET A$=MEMORY$()(16384 TO 23295): REM Запоминаем экран
30 CLS: PRINT "Нажмите любую клавишу"
40 PAUSE 0: POKE 16384, A$
```

Возможно также быстрое копирование участков памяти. Дополним предыдущую программу строками:

```
70 LET A$=MEMORY$ () (18432 TO 20479)
80 POKE 16384, A$: POKE 20480, A$
```

Средняя треть экрана будет скопирована в верхнюю и нижнюю.

ОПЕРАЦИИ С СИМВОЛЬНЫМИ СТРОКАМИ

INSTRING(), STRING$(), SHIFT$(), PRINT #14

INSTRING(<начальная позиция>,<строка символов>,<строка-образец>)

Эта функция осуществляет поиск <строки-образца> в <строке символов>, начиная с позиции, заданной параметром <начальная позиция>. Она возвращает номер позиции в <строке символов>, в которой первый раз встретился образец. Если поиск оказался безуспешным, функция возвратит ноль. Нулевое значение возвращается функцией также:

- если значение параметра <начальная позиция> выходит за пределы базовой строки,
- если <строка-образец> по длине больше исходной <строки символов>, а также
- если одна из строк имеет нулевую длину.

<Строка символов> может иметь произвольную длину, но <строка-образец> не должна состоять более чем из 256 символов.

Для поиска можно использовать шаблон, вставляя в <строку-образец> знак #. В этом случае при поиске содержимое позиции, в которой стоит #, будет игнорироваться. Например, инструкция

```
PRINT INSTRING (1, A$, "baob#b")
```

будет искать в строке A$ позиции строк baobab, baobbb, baobcb и т. д.

INSTRING в сочетании с функцией MEMORY$ (см. предыдущий раздел) позволяет вести поиск в обширных участках памяти компьютера.

STRING$(<число>,<строка символов>)

Функция возвращает символьное значение, состоящее из <число> раз повторенных <строк символов>. Так, например, STRING$(5,"la-") сформирует строку "la-la-la-la-la-".

Функция STRING$ формирует строку быстрее, чем цикл FOR...NEXT, и поэтому ее целесообразно использовать совместно с оператором POKE для заполнения памяти требуемыми данными. К примеру, для быстрой очистки трети экрана достаточно выполнить строку:

```
POKE 16384, STRING$(2048,CHR$ 0)
```

Пригодится функция STRING$ и для формирования шаблона в операторе USING, когда число печатаемых разрядов определяется значениями переменных, скажем, переменных Intr и Fr в следующем примере:

```
PRINT USING STRING$(Intr,"#")+"."+STRING$(Fr,"#"); A
```

SHIFT$ (<номер режима>,<строка символов>)

Эта функция в зависимости от выбранного <номера режима> (1...7) преобразует <строку символов>: избирательно заменяет символы другими символами, последовательностями символов или удаляет символы из строки.

Режим 1. В этом режиме функция SHIFT$ заменяет все прописные буквы в <строке символов> на строчные.

Режим 2. Обратная процедура: заменяет строчные буквы прописными.

Режим 3. Все прописные буквы заменяются на строчные, а строчные — на прописные.

Режим 4. Функция возвращает строку, в которой все управляющие символы (с кодами от 0 до 31), за исключением символа возврата каретки (код 13), заменены символом точки (CHR$ 46)*. Используя этот режим, удобно просматривать память компьютера, например, область бейсик-программы:

```
PRINT SHIFT$(4, MEMORY$()(<адрес начала> TO <адрес конца>))
```

Режим 5. Функция SHIFT$ заменяет каждый символ с кодом N из второй половины таблицы символов (ключевые слова интерпретатора) на символ с кодом N–128**. Кроме этого, подобно режиму 4, все управляющие символы (как ранее существующие, за исключением символа возврата каретки, так и полученные в результате преобразования) заменяются символом точки. Заменяется точкой и символ CHR$ 141, который после преобразования должен был бы стать символом возврата каретки (141–128=13).

Режим 6. Целиком совпадает с режимом 5, за исключением одного нюанса: символ возврата каретки уже не находится на особом положении. Он, подобно другим управляющим символам, обращается в точку.

Режим 7. Прежде всего в этом режиме функция SHIFT$ удаляет из строки все символы управления выводом на экран (перемещение курсора, изменение позиции печати, управление цветовыми атрибутами) (см. табл. 19). Причем удаляет не только управляющие символы, но и их параметры*** (если они предусмотрены). Так, к примеру, изымая символ позиционирования вывода на экран — CHR$ 22 (AT-управляющий), функция SHIFT$ «подчистит» и два обязательно следующих за ним параметра. А вот управляющий символ табуляции CHR$ 23 (TAB-управляющий) будет не только удален вместе со своими параметрами, но еще и заменен необходимым количеством пробелов.

---

* См. таблицу символов на стр. 100.

** На уровне двоичного представления чисел это означает, что в любом коде с номером больше 127 сбрасывается старший бит.

*** Функция SHIFT$ в режиме 7 очень «не любит» управляющие символы с параметрами, выходящими за пределы допустимых значений — их наличие в строке может привести к сбросу компьютера.

Таблица 19. **Замена символов функцией SHIFT$ в режиме 7.**

Коды символов	Действие при выводе на экран	Замена
8...11	перемещение курсора	«пустая строка»
12	удаление символа (Delete)	CHR$ 32
15	пробел + возврат каретки + перевод строки	CHR$ 32+CHR$ 13
16...21 + 1 параметр	управление атрибутами при выводе на экран	«пустая строка»
22 + 2 параметра	AT-управляющий	«пустая строка»
23 + 2 параметра	TAB-управляющий	соответствующее число пробелов

Помимо изложенного, функция SHIFT$ в режиме 7 вводит новый управляющий код, который ранее в ZX Spectrum не использовался: в исходной строке символ с кодом 15 она заменяет сочетанием символов пробела (CHR$ 32) и возврата каретки (CHR$ 13).

Но самое ценное в работе функции SHIFT$ в режиме 7 — это замена символов с кодами 128...255 на последовательности букв, составляющих соответствующие ключевые слова интерпретатора. Такая интерпретация выполняется только в режиме клавиатуры KEYWORDS 1. Например, функция SHIFT$(7,CHR$ 140) вернет строку из пяти символов с кодами 76, 79, 67, 65 и 76, которые составят слово LOCAL. Несколько иначе трактуется символ с кодом 168, соответствующий ключевому слову FN. Если двум символам — букве и открывающей скобке или букве и знаку $, стоящим за CHR$ 168, соответствует одна из встроенных функций Beta Basic, то все эти символы будут заменены на полное буквенное название соответствующей функции. В противном случае код 168 будет просто заменен сочетанием символов F и N.

В режиме KEYWORDS 0 никаких изменений в старшей половине кодовой таблицы не происходит, за исключением замены по описанному сценарию символа с кодом 168.

Свойства функции SHIFT$ в режиме 7, к примеру, дают возможность написать на языке Beta Basic драйвер принтера, позволяющего распечатывать тексты программ, или драйвера экрана для независимого редактора программ.

Завершим рассказ о способах обработки символьных значений описанием интереснейшего объекта системы Beta Basic — символьной переменной Z$. Использовать эту переменную можно, как и любые другие (A$...Y$); при обычных операциях с символьными значениями Z$ ничем не отличается от них, но...

Но кроме этого в символьную переменную Z$ можно заносить данные с помощью оператора PRINT (или LPRINT) в виде, предназначенном для вывода на экран. Данные, направленные в Z$, можно

аккумулировать, то есть использовать переменную в качестве своего рода буфера*.

«Печать» данных в переменную Z$ производится оператором PRINT #14. Выполним программу:

```
10 INPUT LINE A$: LET Z$=""
20 PRINT #14; AT 5,10;INK 4;A$;PAPER 2;12345
```

В результате работы программы на экране не появятся ни значение, присвоенное переменной A$, ни числа, стоящие в строке за PRINT — все они будут спрятаны в символьную переменную Z$. В этом легко убедиться, распечатав ее содержимое с помощью оператора PRINT Z$. Данные предстанут на экране, причем будут выведены именно в том цвете и окажутся именно в тех позициях, которые были предусмотрены в операторе PRINT #14. Все это говорит о том, что в переменную Z$ передаются не только данные, но и все коды управления выводом на экран.

Как уже упоминалось, переменная Z$ накапливает выводимую в нее информацию. При этом, если в конце очередного PRINT #14 после выводимых данных не стоит запятая или точка с запятой, они будут дополнены символом CHR$ 13 (возврат каретки). Вся информация может быть мгновенно удалена оператором LET Z$="". Попытка осуществить вывод в Z$ до того, как она была впервые определена, приведет к выдаче сообщения Variable not found.

Описанные свойства символьной переменной Z$ позволяют находить изящные решения многих вопросов программирования. Так, с ее помощью можно задерживать предназначенную для вывода информацию, при желании анализировать или видоизменять ее, пользуясь всеми доступными средствами для работы с символьными переменными, а затем по выбору направлять на то или иное внешнее устройство.

В заключение добавим, что такими «волшебными» свойствами можно наделить любую символьную переменную. Для этого достаточно выполнить оператор

```
POKE 47027,"<имя переменной>"
```

— где <имя переменной> — заглавная латинская буква (без знака $). Так, инструкция POKE 47027,"A" предпишет каналу "Z" работать с переменной A$. Делая аналогичные переключения на разные переменные, можно в любую из них осуществлять запись оператором PRINT #14 со всеми вытекающими отсюда последствиями.

Обращаем внимание на то, что переменные, стоящие в списке вывода оператора PRINT #14, должны быть определены или иметь последние изменения до инициализации переменной Z$. Если же это условие соблюсти невозможно, перед оператором PRINT #14 необходимо делать переинициализацию Z$ (без потери информации в ней) инструкцией LET Z$=Z$.

---

* Для направления печати в переменную Z$ при старте Beta Basic в дополнение к уже существующим каналам ("K", "S", "R" и "P") инициализируется еще один — "Z". К нему по умолчанию подключается поток #14 (см. [1]).

РАБОТА С МАССИВАМИ

SORT, INARRAY(), LENGTH(), COPY, JOIN, DELETE, CHAR$(), NUMBER()

SORT [INVERSE] <символьный массив|числовой массив|строка символов>

Этот оператор упорядочивает элементы числового или символьного массива. Числовой массив должен быть одномерным, символьный — либо одномерным, либо двухмерным (одномерный массив символьных строк). В одномерном символьном массиве оператор SORT расставляет символы (в массиве строк — строки) в алфавитном порядке, точнее, в порядке возрастания кодов. Числовой массив сортируется по значениям его элементов от большего к меньшему. Конструкция SORT INVERSE сортирует элементы в обратном порядке.

SORT позволяет почти мгновенно отыскать максимальное и минимальное значения в числовом массиве:

```
10 DIM A(100)
20 FOR k=1 TO 100: LET A(k)=10*RND: NEXT k
30 SORT A(): LET Max=A(1): LET Min=A(100)
40 PRINT "Min=";Min, "Max=";Max
```



Еще интереснее использовать этот оператор с символьными массивами. Здесь он открывает большие перспективы для создания справочных систем и баз данных.

В символьной строке оператор SORT сортирует символы; в массиве символьных строк переставляет строки, а порядок символов в пределах каждой строки не изменяет. То есть последовательность строк в массиве становится подобной размещению слов в словаре.

Сферу влияния оператора SORT можно ограничить заданным диапазоном строк с помощью сечения массива, например:

```
SORT A$(1 TO 40)
```

Сортировать массив можно не только по первым символам строк, но и по произвольному сечению, заданному во втором измерении. К примеру, строка

```
SORT A$(1 TO 40)(2 TO)
```

отсортирует первые 40 строк массива, игнорируя их первый символ.

INARRAY(<массив строк>,<строка-образец>**)**

По многим свойствам эта функция напоминает функцию INSTRING. Она возвращает номер элемента массива символьных строк, в котором обнаружено первое включение <строки-образца>. На элемент массива (строку), начиная с которой необходимо вести поиск, указывает первый аргумент функции, например:

```
INARRAY(A$(20),"A")
```

Эта функция начнет поиск символа A с 20-го элемента массива (то есть с 20-й строки).

COPY <массив 1> **TO** <массив 2>

JOIN <массив 1> **TO** <массив 2>

Оба оператора добавляют к <массиву 2> элементы <массива 1>. Оператор COPY **копирует** указанные элементы, сохраняя их в исходном массиве, а JOIN **переносит**, не сохраняя оригинал («сшивает» два массива). Поддерживается работа с одно- и двухмерными числовыми и символьными массивами.

Допустимо копировать массивы сами в себя.

Рассмотрим действие этих операторов на примере работы с символьными строками и массивами строк. Операции с числовыми массивами выполняются аналогично. В местах, где наблюдаются некоторые различия, мы сделаем соответствующие оговорки.

```
10 LET A$="QWERTYUIOP": LET B$="qwertyuiop"
20 COPY A$ TO B$
30 PRINT "A$=";A$, "B$=";B$
```

Эта небольшая программка выведет на экран:

A$=QWERTYUIOP B$=qwertyuiopQWERTYUIOP

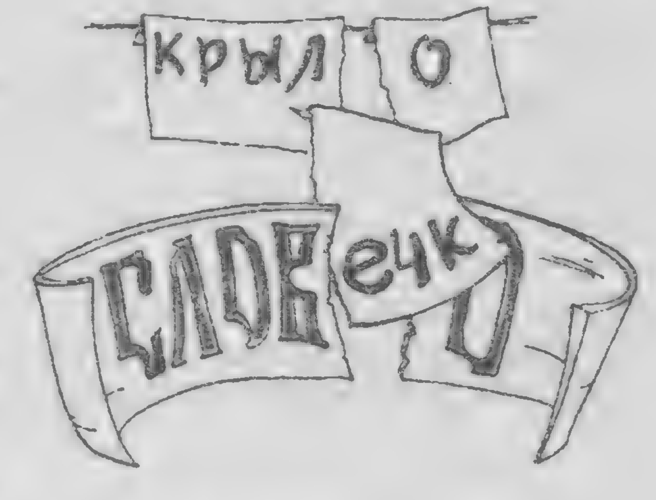
После замены в строке 20 оператора COPY на JOIN выполнение программы прервется на 30-й строке сообщением Variable not found — оператор JOIN ликвидирует переменную A$ после переноса ее содержимого в переменную B$.

Заменим 20-ю строку на

```
20 COPY A$(4 TO 6) TO B$(3)
```

Результат станет таким:

A$=QWERTYUIOP
B$=qwRTYertyuiop



Применение оператора JOIN с такими же параметрами приведет к следующему:

A$=QWEUIOP B$=qwRTYertyuiop

Таким образом, сечение массива-источника (в нашем случае A$(4 TO 6)) задает фрагмент, подлежащий переносу или копированию. Номер элемента, указанный в массиве-приемнике, (B$(3)) определяет место в нем, куда будет вставлен соответствующий фрагмент.

При работе с числовыми массивами также допустимо задавать их сечения, используя тот же синтаксис, что и с символьными массивами.

Операции копирования и переноса двухмерных массивов более сложны. Для их анализа Вы можете написать специальную программу. Мы же ограничимся приведением диаграмм, иллюстрирующих работу операторов COPY и JOIN с массивами разных размеров и размерностей (рис. 24). На диаграммах представлены

массивы A() и B() (они могут быть как числовыми, так и символьными), а также результаты выполнения оператора COPY A() TO B() или JOIN A() TO B().

```
1) |BBB|          |BBB|        2) |BBBBB|         |BBBBB|
   |BBB|          |BBB|           |BBBBB|         |BBBBB|
              ->  |AAA|                       ->  |AAAoo|
   |AAAA|         |AAA|           |AAA|           |AAAoo|
   |AAAA|                         |AAA|

3) |B|            |B|          4) |BBBBB|         |BBBBB|
   |B|            |B|             |BBBBB|         |BBBBB|
   |B|        ->  |B|                         ->  |Aoooo|
                  |A|             |A|             |Aoooo|
   |AAAA|         |A|             |A|             |Aoooo|
   |AAAA|                         |A|
```

A и *B* — элементы соответствующих массивов;
o — нулевое значение в числовом массиве или пробел — в символьном.

Рис. 24. Работа COPY и JOIN с двухмерными массивами.

Из этих диаграмм становится ясно, что в случае, когда второе (либо единственное) измерение массива-приемника меньше второго (либо единственного) измерения массива-источника, массив-приемник играет роль «прокрустова ложа» — пристраиваемый фрагмент или весь массив будет усечен по второй размерности до границ своего нового пристанища (случаи 1 и 3).

Если же в новом жилище исходному массиву или его фрагменту слишком просторно (второе измерение массива-приемника больше, чем массива-источника), в этих случаях новые строки символьного массива дополнятся пробелами. В числовых же массивах вместо недостающих элементов будут стоять нули (случаи 2 и 4).

Для всех вариантов применения оператора JOIN, если не было указано сечение массива-источника, последний будет удален из области переменных.

Синтаксис описываемых операторов допускает использование сечений только по первому измерению.

Как и случае одномерных массивов и символьных строк, двухмерные массивы можно копировать или переносить не только в конец массива-приемника, но и в любое другое его место.

DELETE <массив|символьная переменная>

Оператор удаляет из области переменных символьный или числовой массив либо, если задано сечение массива, его фрагмент. Он также работает с символьными и числовыми переменными,

удаляя их из области переменных. Действие оператора DELETE демонстрирует следующая программа:

```
10 DIM A(30,100)
20 DELETE A(11 TO 20)
30 PRINT "A(";LENGTH(1,"A()");",";LENGTH(2,"A()");")"
```

Она напечатает: A(20,100), то есть оператор DELETE при указании сечения сокращает массив по первому измерению. В случае работы с символьными строками он позволяет вырезать из них заданные фрагменты. При этом оператор DELETE C$(M TO N) равносилен строке LET C$=C$(TO M)+C$(N TO).

Удаляя из программы переменные и массивы, отслужившие свой век, DELETE позволяет экономно расходовать память компьютера.

LENGTH(<измерение массива>,"<имя массива>")

Функция LENGTH определяет количество элементов одно- или двухмерного массива (как символьного, так и числового) по каждому его измерению. Номер измерения задается первым аргументом функции, который может принимать значения 1 или 2. Имя массива заключается в кавычки*:

```
10 DIM A(10,45)
20 PRINT LENGTH(1,"A()"), LENGTH(2,"A()")
```

В результате работы этого примера будут выведены два числа — 10 и 45.

Функция LENGTH позволяет работать с массивом, размеры которого в данной точке программы неизвестны, без опасения выйти за его пределы. Это может понадобиться в процедурах, где универсальный алгоритм рассчитан на работу с массивами, в общем случае имеющими разную длину. Например, программа сортировки элементов одномерного массива (см. стр. 273) может быть переписана так:

```
  10 DIM G(100)
  20 FOR k=1 TO 100
  30 LET G(k)=10*RND
  40 NEXT k
  50 PROC Limits G(),Emax,Emin
  60 PRINT "Min=";Emin, "Max=";Emax
1000 DEF PROC Limits REF F(), REF Max, REF Min
1010 LOCAL Lng,Q()
1020 LET Lng=LENGTH(1,"F()")
1030 DIM Q(1)
1040 COPY F() TO Q(): DELETE Q(1)
1070 SORT INVERSE Q()
1080 LET Min=Q(1): LET Max=Q(Lng)
1090 END PROC
```

---

* Надо отметить, что в LENGTH допустимо подставлять имя массива как в форме "G()"; так и в виде имени любого из его элементов: "G(k)" или "G(<число>)". На работе функции эти различия в записи имени не отражаются.

В начале этой программы при помощи генератора случайных чисел формируется массив G() (причем размер массива может быть произвольным), и ссылка на него передается в процедуру Limits (строка 50). В ней вычисляется размер массива F() (процедура «знает» массив G() только под этим именем; строка 1020) и затем его содержимое копируется в массив Q() (строка 1040), объявленный ранее как локальный (строка 1010). Далее идет сортировка (строка 1070) и нахождение предельных значений (строка 1080) по старой схеме. По выходу из процедуры не определенные ранее переменные Emax и Emin будут иметь значения, которые получили в процедуре их alter ego — переменные Max и Min.

В завершение этого раздела мы расскажем о двух функциях, также позволяющих значительно экономить память при работе с целочисленными массивами:

CHAR$(<число>)

NUMBER(<двухсимвольная переменная>)

Как известно, под хранение **любого** числа в памяти ZX Spectrum отводится 5 байтов, в то время как для представления **целого** числа из интервала 0...65535 достаточно только двух. Поэтому целочисленные массивы преимущественно «хранят воздух» — на 3/5 они заполнены нулями. Бороться с такой непозволительной роскошью помогает функция CHAR$: она представляет числа из интервала 0...65535 двухбайтовыми строками, то есть равносильна следующему набору инструкций:

```
LET A=INT (N/256): LET B=N–A*256
LET D$=CHR$ A+CHR$ B
```

– где N — целое число из указанного диапазона.

Функция NUMBER() производит обратную операцию — переводит двухсимвольные строки в форму целых чисел. Аналогом функции является строка:

```
LET N=256*CODE D$(1)+CODE D$(2)
```

Ниже мы приводим пример, демонстрирующий экономию памяти, достигаемую с помощью этих функций:

```
10 DIM D$(500,2)
20 FOR K=1 TO 500
30 LET D$(K)=CHAR$(K): NEXT K
40 PRINT MEM()
50 DIM N(500)
60 FOR K=1 TO 500
70 LET N(K)=NUMBER(D$(K))
80 NEXT K: DELETE D$()
90 PRINT MEM()
```

В первом цикле программы с помощью функции CHAR$ из целых чисел от 1 до 500 формируется символьный массив (строки 10...30), после чего на экран выводится количество памяти в байтах, доступной для бейсик-программ (строка 40). Во втором цикле (строки 60...80) функция NUMBER переводит символьный массив в

числовой. После удаления символьного массива снова печатается объем свободной памяти. Анализируя результат, можно сделать вывод, что для целочисленного массива размером лишь в 500 элементов преобразование его в двухбайтовое представление дает экономию в полтора килобайта.

РАБОТА С КЛАВИАТУРОЙ

EDIT, GET

Помимо описанных выше средств программного преобразования символьных строк, Beta Basic позволяет вручную редактировать строковые, а также и числовые данные. Делается это с помощью оператора EDIT, уже знакомого по описанию редактора Beta Basic. В новом амплуа он имеет следующий синтаксис:

EDIT [LINE] <символьная переменная>|<числовая переменная>

По действию оператор EDIT близок к INPUT — он присваивает переменной, указанной следом за ним, соответственно символьное или числовое значение, введенное с клавиатуры. Но в отличие от INPUT, он не отнимает у переменной ее прежнего содержимого, а выводит его в строку служебного экрана для редактирования. Вот небольшой пример:

```
10 LET A$="QWERTY", B$="qwerty"
20 INPUT "A$="; LINE A$,
30 EDIT "B$="; LINE B$
40 PRINT "A$=";A$,"B$=";B$
```

Если при выполнении этой программки на запросы обоих операторов просто нажать клавишу Enter, то PRINT (строка 40) «покажет», что переменная A$ «опустошилась», а B$ сохранила ранее присвоенное ей значение. Поместив перед именем переменной точку с запятой, можно организовать редактирование и числовых переменных. Если стоящая в EDIT переменная не была к этому моменту определена, то оператор EDIT будет работать как обычный INPUT. В списке из нескольких переменных для всех из них, кроме первой, EDIT тоже работает как INPUT.

Недопустимо подсовывать оператору EDIT «пустую строку» — это может привести к сбросу или зависанию компьютера. Поэтому перед применением EDIT в программе (если нет уверенности) рекомендуется делать проверку переменной на «пустоту»:

```
IF A$="" THEN INPUT A$: ELSE EDIT A$
```

GET <символьная переменная|числовая переменная>

Оператор GET производит опрос клавиатуры и присваивает переменной символьное или числовое (в зависимости от того, какая переменная используется в качестве параметра) значение, соответствующее нажатой клавише, одной или в комбинации с функциональными клавишами CS и SS. На нажатие самих функциональных клавиш GET не реагирует. В отличие от функции

стандартного Бейсика INKEY$, оператор GET задерживает выполнение программы и ждет, пока не будет нажата клавиша. Он в некоторой степени аналогичен комбинации

```
PAUSE 0: LET A$=INKEY$
```

Кроме того, оператор GET позволяет менять режимы курсора во время ожидания нажатия клавиши: он не реагирует на нажатие комбинаций клавиш, задающих тот или иной тип курсора. Используя это свойство, можно считывать символы, возвращаемые клавиатурой не только в режимах курсора [L] или [C], как при работе с функцией INKEY$, но и в режимах [E] и [G].

Оператор GET с числовой переменной в качестве параметра при нажатии цифровых клавиш в режимах курсора [L] и [C] присваивает ей, соответственно, значения от 0 до 9. При нажатии клавиши A переменной присваивается значение 10, B — 11 и т. д. по алфавиту.

Рассмотрим небольшой пример:

```
10 PRINT OVER 1;"_";CHR$ 8;: GET A$
30 PRINT OVER 1;"_";CHR$ 8;: 40 PRINT A$;
50 GO TO 10
```

Для такой крохотной программки результат неожиданный и впечатляющий: теперь можно свободно перемещать курсор по экрану и вводить символы и ключевые слова в любом его месте. Чтобы избежать сообщения Invalid colour при попытке удалить символ клавишами CS/0, находясь в режиме курсора [E]*, целесообразно заменить 40-ю строку программы на

```
40 IF CODE A$<16 OR CODE A$>31 THEN PRINT A$
```

Согласитесь, получившиеся пять строк — неплохая заготовка для создания полноэкранного редактора программ Beta Basic. Но вот проблема: как включить введенную строку в текст программы? Beta Basic имеет решение и на этот случай: в нем предусмотрен оператор KEYIN, который мы рассмотрим чуть позже.

РАБОТА СО СПРАЙТАМИ

GET, PLOT, CHR$ 0, CHR$ 1

Способности оператора GET не ограничены тем, что мы описали в предыдущем разделе. Этот оператор предоставляет также возможность работать со спрайтами, позволяя получать неплохие результаты в области мультипликации.

* Режим [E] в данном случае не отключается автоматически после ввода одного символа, как это происходит в редакторе; возврат к курсорам [L] или [C] необходимо делать принудительно повторным нажатием CS/SS.

Формат оператора таков:

GET <строковая переменная>, <горизонтальная координата>, <вертикальная координата> [, <ширина>, <длина>] [;<тип>]

Действие оператора элементарно: он помещает в <строковую переменную> («фотографирует») прямоугольную область экрана. <Ширина> и <длина> этой области измеряется в знакоместах, <координаты> верхнего левого угла — в пикселях. Сделать «отпечаток» с этой области можно, выводя <строковую переменную> операторами PRINT или PLOT в любое место экрана. При этом с помощью CSIZE возможно изменение масштаба изображения (CSIZE в нашем «фотопроцессе» играет роль увеличителя). Аналогия с фотографией на этом не исчерпывается: значения параметра <тип> определяют, будет ли наша «фотография» черно-белой или цветной. Нулевое значение параметра указывает на то, что запомнить следует только изображение, а единица — изображение с атрибутами. По умолчанию параметры принимают следующие значения: <ширина>=<длина>=1, <тип>=0. Отметим, что установка CSIZE 0 не позволит Вам воспроизводить картинку с помощью PRINT. Перейдем к примеру.

```
 5 CSIZE 8
10 PRINT "BOMM!"''''"BAMM!": REM Вывели две надписи
20 CIRCLE 16, 159, 5: REM В промежутке нарисовали окружность
30 GET A$, 0, 175, 5, 4; 0: REM «Сняли» надписи вместе с окруж-
   ностью
40 PRINT AT 0,0; OVER 1; A$: REM Стираем нашу картинку
50 FOR I=1 TO 150: REM Вновь и вновь, двигаясь по диагонали...
60 PLOT I, 175—I, A$: REM ...воспроизводим картинку...
70 OVER 1: PLOT I, 175—I, A$:OVER 0:REM ...и тут же снова ее
   стираем
80 NEXT I
```

Ну, чем не мультипликация!

Мы убедились, что оператору GET «по зубам» и символы, и графика. Остается выяснить, что же представляет собой полученная нами переменная A$. Это может понадобиться для конструирования спрайтов «в уме», без помощи оператора GET, что, как мы покажем, может дать определенные удобства.

Первым элементом такой переменной всегда является управляющий символ CHR$ 0 или CHR$ 1 и это сразу выделяет спрайты из числа обычных символьных переменных. Код 0 говорит о том, что за ним следуют 8 байт, которые должны интерпретироваться как закодированное изображение символа 8×8 пикселей (аналогично тому, как кодируются символы UDG). Код 1 предупреждает интерпретатор о том, что эти 8 байт предваряются еще одним, содержащим атрибуты знакоместа. Таким образом, эти коды совпадают со значением параметра <тип> оператора GET. Каждое последующее знакоместо (если их больше одного) представлено таким же блоком из управляющего кода 0 или 1 и следующих за ним 8 или 9 байт. При необходимости между блоками располагаются коды управления курсором (символы CHR$ 8...11, стр. 245), которые задают взаимное расположение знакомест. Попробуйте про-

анализировать содержимое переменной A$, полученной в приведенном примере, и «анатомия» спрайтов будет для Вас понятнее.

Однократное применение GET не позволит Вам получить спрайт с формой, отличающейся от прямоугольной. Однако «склеив» (с помощью плюса или JOIN) несколько строковых переменных, «разбавляя» их кодами управления курсором, можно получить «суперспрайт» самой экзотической формы — даже состоящий из несвязанных фрагментов. Фантастические возможности появляются при объединении в одну переменную обычных текстовых строк и строк-спрайтов, а также спрайтов типов 0 и 1! Предоставим Вам возможность поэкспериментировать самостоятельно.

ГЕНЕРАЦИЯ ДАННЫХ И ТЕКСТА ПРОГРАММ

KEYIN

KEYIN <строка символов>

Оператор KEYIN помещает <символьную строку> в текст программы. Если строка имеет номер и не противоречит соглашениям синтаксиса Beta Basic, она займет соответствующее место в программе. В противном случае компьютер выдаст сообщение Nonsense in Basic. Выполним строку программы:

```
10 KEYIN "20 PRINT ""Hello"""
```

Она выведет на экран:

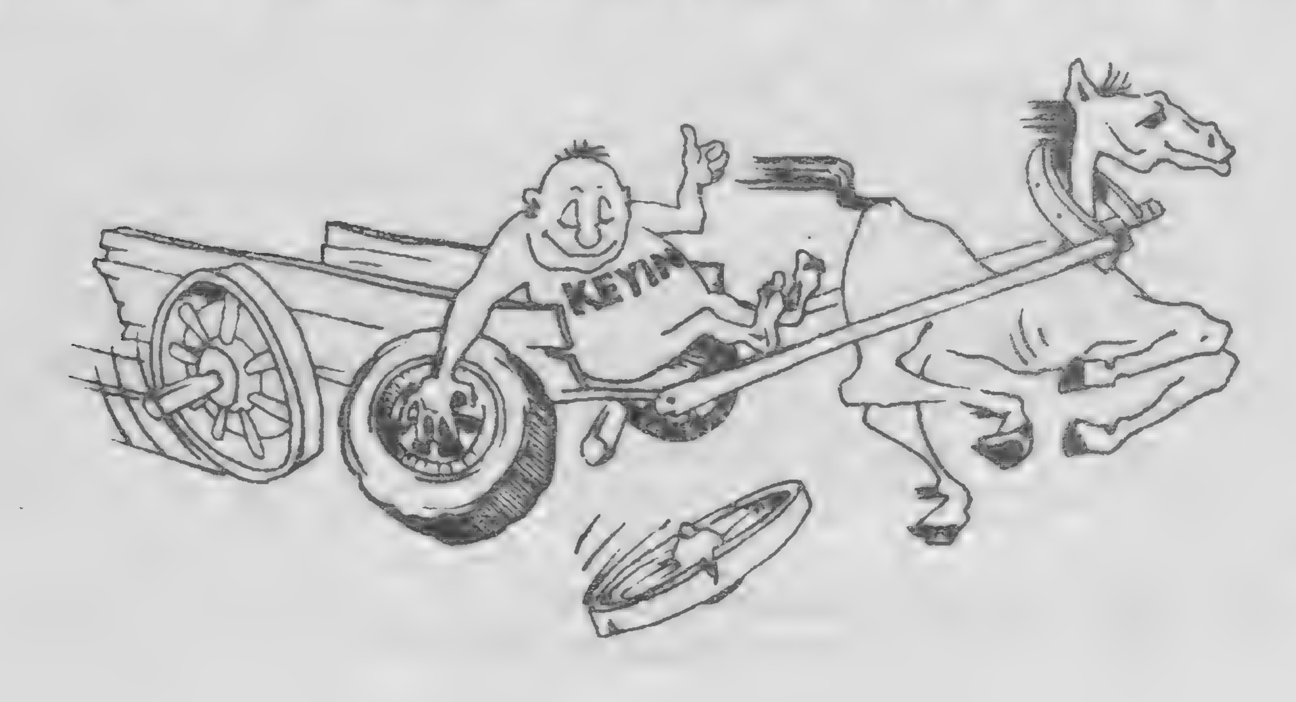
Hello

— а текст примера станет выглядеть так:

```
10 KEYIN "20 PRINT ""Hello"""
20 PRINT "Hello"
```

Таким образом, описываемый оператор позволяет программе модифицировать саму себя по ходу выполнения!

При работе оператора KEYIN могут встретиться некоторые неожиданности. Отметим лишь, что KEYIN гораздо надежнее работает,



когда генерируемые им строки располагаются в тексте программы после него.

Сфера применения оператора KEYIN весьма широка. В первую очередь необходимо сказать о возможности с его помощью генерировать данные для DATA-списков. Такая потребность может возникнуть, если необходимо передать данные из одной программы, работающей в среде Beta Basic, в другую, которая для ускорения расчетов должна быть скомпилирована. Как известно, компиляторы Spectrum-Бейсика не позволяют работать «в лоб» с внешней памятью (магнитофоном или дисководом). При помощи же оператора KEYIN данные, сформированные первой программой и сохраненные в виде массива на ленте, можно преобразовать в списки операторов DATA. Полученные DATA-списки объединяются со второй программой и совместно с ней компилируются.

ПРЕОБРАЗОВАНИЕ ЧИСЕЛ

BIN$(), HEX$(), DEC()

Spectrum-Бейсик располагает только одной функцией преобразования чисел из одной системы счисления в другую — BIN, переводящей двоичные числа в десятичные. Beta Basic добавляет еще три: BIN$, HEX$ и DEC.

Функция

BIN$(<число>)

выполняет действие, обратное функции BIN — переводит числа в интервале 0...65535 из десятичного в двоичное представление. Результат возвращается в виде символьной строки длиной 8 символов для чисел в интервале 0...255 и 16 символов — для остальных чисел из допустимого диапазона. Наглядно работу функции представляет следующий пример:

```
DO: PRINT AT 0,0; BIN$(IN 65022): LOOP
```

Порт 65022 обслуживает клавиатуру, точнее, клавиши A, S, D, F и G (подробнее см. в [1]). Если эти клавиши не нажимать, то на экране будет представлено число ×××11111 (5, 6 и 7-й биты предназначены для других целей и могут принимать произвольное значение). Нажатие перечисленных клавиш приведет к появлению нулей в соответствующих позициях числа. Так, удерживаемые одновременно клавиши A и G дадут такой результат: ×××01110.

С помощью функции BIN$ удобно просматривать область символов, определяемых пользователем, область атрибутов, системные переменные и т. п. Для получения в результирующей строке символов, отличных от 0 или 1 (например, пробела и черного квадрата из набора псевдографических символов ZX Spectrum), необходимо выполнить

```
POKE 62869,"<символ, заменяющий ноль>":
POKE 62865,"<символ, заменяющий единицу>"
```

Функция

HEX$(<число>)

переводит десятичное целое число (например, адрес ячейки памяти) в шестнадцатеричный вид. Результат возвращается в виде двухсимвольной строки для чисел от –255 до 255 и четырехсимвольной — для всех больших из диапазона –65535...65535.

Операцию обратного преобразования выполняет функция

DEC(<строка символов>)

Она переводит числа из шестнадцатеричного представления в десятичное. Аргумент функции должен быть двух- или четырехсимвольной строкой, представляющей собой шестнадцатеричное число (шестнадцатеричные цифры A...F могут быть как строчные, так и прописные). Например, в результате выполнения оператора PRINT DEC("9C40") на экран будет выведено число 40000.

МАТЕМАТИЧЕСКИЕ ФУНКЦИИ

COSE(), SINE(), RNDM(), AND(), OR(), XOR(), MOD()

Язык программирования Beta Basic вводит несколько новых математических функций и среди них две тригонометрические (аналоги уже существующих в Spectrum-Бейсике):

COSE(<число>)
SINE(<число>)

Появление этих дубликатов оправдано тем, что они работают в пять раз быстрее своих предшественников, правда, с несколько меньшей точностью (вследствие четырехбайтового представления обрабатываемых ими чисел), что, впрочем, в большинстве случаев допустимо.

Функция

RNDM(<число>)

возвращает случайное число в интервале от 1 до числа, заданного аргументом. Помимо того, что RNDM работает с вдвое большей скоростью относительно стандартной RND, она генерирует последовательность псевдослучайных чисел с менее заметной закономерностью. Действительно, строка

```
DO: PLOT RND*255,RND*175: LOOP
```

– покрывая экран точками со случайными координатами, уже на пятой минуте своей работы обнаружит явную тенденцию размещать их с определенной закономерностью, а через полчаса просто покроет экран регулярной сеткой из ломаных линий. После замены этой строки на

```
DO: PLOT RNDM (255),RNDM (175): LOOP
```

экран будет покрываться точками равномерно по всей поверхности.

Три новые функции: AND, OR и XOR выполняют операции бинарной математики. Функция

AND(<число>,<число>)

осуществляет поразрядное логическое умножение (операцию "И") двух чисел, находящихся в интервале 0...65535. Одно из применений функции AND — это определение одновременного нажатия клавиш при опросе клавиатуры, как это делается в следующем примере:

```
 90 IF AND(BIN 00010001,IN 65022)=0 THEN BEEP 0.1,12
100 GO TO 90
```

Значение, считанное из порта 65022 (клавиши **A**, **S**, **D**, **F** и **G**), маскируется двоичным числом 00010001. Если на клавиатуре нажать одновременно клавиши **A** и **G**, из порта будет считано число 11101110, и только в этом случае функция AND вернет ноль и программа подаст звуковой сигнал.

Функция

OR(<число>,<число>)

делает поразрядное логическое сложение (операцию "ИЛИ") двух чисел (0...65535).

С помощью функции OR можно, например, определить, не нажата ли хотя бы одна из интересующих нас клавиш. Для клавиш **A** или **G** решение поставленной задачи выглядит так:

```
90 IF OR(BIN 11101110,IN 65022)<>255 THEN BEEP .1,12
```

Функция

XOR(<число>,<число>)

выполняет операцию "исключающее ИЛИ" двух чисел (0...65535).

Еще одна математическая функция:

MOD(<число 1>,<число 2>)

возвращает остаток от целочисленного деления <числа 1> на <число 2>. Оба аргумента должны быть положительными. Например, MOD(29,9) возвращает число 2, поскольку 29−9×3=2.

ПРОЧИЕ ОПЕРАТОРЫ И ФУНКЦИИ

Beta Basic расширяет возможности оператора SAVE, сохраняя его исконное назначение — записывать программы и данные на магнитную ленту. Прежде он записывал программу только целиком. Beta Basic же допускает новый синтаксис этого оператора:

SAVE <номер строки> **TO** <номер строки>[;<логический адрес устройства>];|,<"имя файла"> [**LINE** <номер строки автостарта>]

Этот синтаксис обеспечивает сохранение фрагмента программы. Так, оператор SAVE 10 TO 120;"TEST" запишет на магнитофон часть

программы с 10-й строки по 120-ю включительно под именем TEST. При этом область переменных на ленте не сохраняется. Команда SAVE DATA <"имя файла"> сохранит на внешнем носителе только область переменных программы.

В связи с новыми возможностями SAVE закономерно появление оператора

VERIFY <номер строки> **TO** <номер строки>
[;<логический адрес устройства>];|,<"имя файла">

Очевидно, что он проверяет правильность записи на ленте фрагмента программы, ограниченного указанными номерами строк.

Использование параметра <логический адрес устройства> (0, 1, 2...) имеет смысл только при подключенном интерфейсе типа Interface 1, поддерживающем микродрайвы, локальную сеть и т. д.. (см. [1]). Имя устройства, с которым будут работать операторы LOAD, SAVE, VERIFY и MERGE можно задавать и по умолчанию, используя уже знакомый нам оператор DEFAULT. В этом случае он имеет следующий формат:

DEFAULT=<имя устройства>

Здесь под <именем устройства> понимаются логические имена внешних устройств: t (магнитофон); m<номер микродрайва> (микродрайв); n<номер узла> (узел локальной сети); b (интерфейс RS-232C, канал "B"); d (диск).

Наполняется новым содержанием и оператор CLEAR. Выполнение его с параметром в интервале −767...767 (3×256−1) смещает RAMTOP вверх или вниз на заданное число байт. Заметим, что при выполнении этого оператора не делается проверка на пересечение RAMPTOP с текстом программы.

Интерпретатор Beta Basic работает во 2-м режиме прерываний (IM2) процессора Z80. Благодаря этому возможен принудительный выход в интерпретатор из программ в машинных кодах (не изменяющих режим прерываний). Для этого необходимо удерживать клавишу **Break** (**CS/Space**) более одной секунды. Именно таким образом прерывается работа операторов REF и AUTO. Иногда такой прием позволяет вырваться из зависшего состояния компьютера. К сожалению, 2-й режим прерываний не позволяет «в лоб» работать с операционной системой TR-DOS. Тем не менее, существует несколько приемов, позволяющих обойти эти трудности. Они описаны в Приложении 5.

На этом закончим рассказ об операторах и функциях интерпретатора Beta Basic. Хотим только отметить, что мы ни в коей мере не претендуем на полноту описания этого языка программирования. Не исключено, что некоторые его возможности так и остались неразведанными — излагался лишь наш собственный опыт постижения Beta Basic.

ПРИЛОЖЕНИЯ

1. Алфавитный перечень операторов Beta Basic

Таблица 20.

Ключевое слово, набор	Действие	Пример	Стр.
ALTER [G] A	Поиск и замена по образцу имен переменных, числовых констант, строк символов в тексте программы	ALTER Nm TO Num ALTER 1.1 TO 1.2 ALTER "Bitte" TO "Please"	241
	Смена атрибутов экрана на заданные	ALTER INK 3 TO PAPER 7	249
AUTO [G] 6	Задает режим автоматической генерации номеров строк программы	AUTO 100,10 AUTO 0	239
CLEAR [K] X	Перемещает RAMTOP на указанное количество байтов	CLEAR 15	285
CLOCK [G] C	Устанавливает текущее время; задает время подачи звукового сигнала и/или перехода к подпрограмме; задает подпрограмму обработки прерывания, устанавливает режимы работы часов	CLOCK "15:30:00" CLOCK "A15:35" CLOCK 1200 CLOCK 7	258
CLS [K] V	Стирает содержимое окна с заданным номером	CLS 3	251
COPY [K] Z	Копирует содержимое одного массива или его части в другой массив	COPY A(2 TO 5) TO B(7) COPY A$(TO 6) TO B$(9)	274
CSIZE [G] SS/8	Задает размер символов в текущем окне, переключает драйверы вывода на экран	CSIZE 5,9 CSIZE 0	245
DEFAULT [G] SS/2	Присваивает значение переменной, если она ранее не была определена	DEFAULT A=1, B=2, C=3	244
	Задает по умолчанию устройство ввода-вывода	DEFAULT=t	285
DEF KEY [G] SS/1	Приписывает указанной клавише заданную последовательность операторов	DEF KEY "h";"HELLO:" DEF KEY "8": CSIZE 4,8	239

Ключевое слово, набор	Действие	Пример	Стр.
DEF PROC [G] 1	Заголовок процедуры	DEF PROC Test A,B,REF C DEF PROC Sum DATA	261
DELETE [G] 7	Удаляет фрагмент программы	DELETE 1200 TO 2500 DELETE 0 TO 0	243
	Удаляет массив или часть массива; удаляет часть символьной переменной	DELETE A(15 TO 23) DELETE D$(13)	275
DO [G] D	Заголовок цикла типа DO...LOOP	DO: PLOT RND*255,RND*175: LOOP	265
DPOKE [G] P	Записывает по указанному адресу двухбайтовое число	DPOKE 42484, 5088	268
DRAW TO [K] W + SS/F	Проводит линию или дугу из текущей точки в точку с указанными абсолютными координатами	DRAW TO 70,100 DRAW TO 120,120,3/2*PI	251
EDIT Enter+0	Вызывает на редактирование строку программы	EDIT 30	240
EDIT [G] SS/5	Вызывает на редактирование переменную	EDIT A$ EDIT ;Q	278
ELSE [G] E	Начинает блок операторов в конструкции типа IF...THEN...ELSE, выполняющийся в случае, если условие ложно	IF A=1 THEN: PRINT "By": RETURN: ELSE: PRINT "We'll rock you": GO TO 1282	254
END PROC [G] 3	Оператор конца процедуры. Осуществляет возврат управления основной программе. Аналогичен RETURN	DEF PROC Wiper: CLS 1: IF=1 THEN RETURN: CLS 2: END PROC	261
EXIT IF [G] I	Осуществляет выход по условию из цикла типа DO...LOOP	EXIT IF I=28	266
FILL [G] F	Заливает участок экрана, ограниченный замкнутым контуром, цветом PAPER или INK	FILL INK 3; 70,70	252
GET [G] G	Присваивает символьной переменной символ нажатой клавиши или числовой переменной — порядковый номер клавиши.	GET A$ GET A	278

Ключевое слово, набор	Действие	Пример	Стр.
GET [G] G	Считывает с экрана в символьную переменную спрайт с указанными размерами	GET A$,42,38,2,3	279
GO TO ON [K] G +[G] O	Осуществляет переход на строку из указанного списка в соответствии со значением заданной переменной (выражением)	GO TO ON J; 50,70,90	254
GO SUB ON [K] H +[G] O	Осуществляет вызов подпрограммы, начинающейся со строки из указанного списка, в соответствии со значением заданной переменной (выражением)	GO SUB ON J; 70,80,90	255
JOIN [G] SS/6	Сцепляет строку программы с указанным номером или текущую строку со следующей за ней строкой, записывая их через двоеточие под одним номером	JOIN 50	241
	Переносит содержимое одного массива или его части в другой массив	JOIN A(3 TO 7) TO B(10) JOIN C() TO D() JOIN E$() TO F$(5)	274
KEYIN [G] SS/4	Помещает заданную строку символов в текст программы	KEYIN "1200 LET A=8763"	281
KEYWORDS [G] 8	Переключает режим ключевых слов Beta Basic в режим Graphics и обратно, устанавливает способ ввода ключевых слов в строку редактирования	KEYWORDS 1 KEYWORDS 4	236
LIST [K] K	Выводит на экран (или в символьную переменную Z$ по LIST #14) листинг программы в пределах указанных строк	LIST 10 TO 50 LIST TO 70 LIST #14; 50 TO LIST	243
	Распечатывает текст указанной процедуры	LIST PROC PortRead	261
	Распечатывает номера строк,содержащих указанный ключ	LIST REF "Org" LIST REF Num	243

Ключевое слово, набор	Действие	Пример	Стр.
LIST [K] K	Распечатывает содержимое переменных и размерности массивов программы	LIST DATA LIST VAL LIST VAL$	243
	Распечатывает макроопределения клавиш	LIST DEF KEY	243
LOCAL [G] SS/3	Объявляет переменные процедуры локальными	LOCAL A,B,C	262
LOOP [G] L	Оператор окончания цикла типа DO...LOOP	см. оператор DO	265
ON [G] O	Исполняет один из операторов, стоящих после него в строке, в соответствии со значением указанной переменной (выражения)	ON J: PRINT "Line 1": PRINT "Line 2": PRINT "Line 3"	255
ON ERROR [G] N	Задает номер строки, с которой начинается подпрограмма обработки ошибок, или описывает эту подпрограмму	ON ERROR 3500 ON ERROR: BEEP 1,1: RETURN	255
PLOT [K] Q	Выводит строку символов (спрайт) в заданную точку экрана	PLOT 100,70;"Demo"	246
POKE [K] O	Записывает в память компьютера строку кодов символов, начиная с заданного адреса	POKE 40000, "String"	268
POP [G] Q	Снимает со стека GO SUB и присваивает переменной (если она указана) значение адреса возврата из процедуры или адрес заголовка цикла типа DO...LOOP	POP A	267
PROC [G] 2	Вызывает процедуру с заданным именем и передает ей значения фактических параметров	PROC Sub1 3, , A+2, D(5)	261
READ [E] A	Считывает значения фактических параметров, передаваемых в процедуру списком произвольной длины	DEF PROC Test DATA: DO UNTIL ITEM()=0: READ LINE a$	263

Ключевое слово, набор	Действие	Пример	Стр.
REF [G] SS/7	Осуществляет поиск по образцу в тексте программы и вызывает на редактирование строки, удовлетворяющие условию поиска	REF Num REF "Sub1" REF 3 REF (A$)	240
	задает передачу параметров в процедуру по ссылке	DEF PROC Test REF a	264
RENUM [G] 4	Перенумеровывает строки программы из указанного диапазона, начиная с заданного номера, и с заданным шагом	RENUM 70 LINE 90 STEP 5	242
ROLL [G] R	Осуществляет циклический скроллинг окна экрана на заданное количество пикселей в указанном направлении	ROLL 5;0,175;50,88	252
SAVE [K] S	Производит запись на ленту фрагмента программы или области переменных	SAVE 10 TO 70; "Filename" SAVE DATA "Filename"	285
SCROLL [G] S	Осуществляет скроллинг окна экрана на заданное количество пикселей в указанном направлении	SCROLL 5;0,175;50,88	252
SORT [G] M	Сортирует числовой или символьный массив по убыванию или возрастанию значений его элементов	SORT A() SORT A$(TO 25)(2 TO) SORT INVERSE A()	273
TRACE [G] T	Задает номер строки подпрограммы обработки прерывания, обращение к которой будет осуществляться после каждого выполненного оператора программы или описывает эту подпрограмму	TRACE 7000 TRACE: LIST VAL: RETURN	257
UNTIL [G] K	Задает критерий окончания цикла типа DO...LOOP	DO UNTIL ABS X<Eps	266
USING [G] U	В операторе PRINT (LPRINT) задает шаблон вывода чисел	PRINT USING "#.##"; A	248
VERIFY [E] SS/R	Осуществляет проверку записанного на ленту фрагмента программы из указанного диапазона строк	VERIFY 10 TO 70; "Name"	285

Ключевое слово, набор	Действие	Пример	Стр.
WHILE [G] J	Задает критерий продолжения цикла типа DO...LOOP	LOOP WHILE Y<>3	266
WINDOW [G] 5	Задает координаты и размеры текстового окна с указанным номером	WINDOW 5; 0,175,9,50	249
	Объявляет текущим окно с заданным номером	WINDOW 5	

2. Алфавитный перечень функций Beta Basic*

Таблица 21.

Функция, набор	Возвращаемое значение	Пример	Стр.
AND A	Результат поразрядного "И" двух чисел	AND(A,B)	284
**BIN$** B$	Двоичный эквивалент десятичного числа	BIN$(12)	282
**CHAR$** C$	Представление целого числа из диапазона 0...65535 двумя байтами	CHAR$(40000)	277
COSE C	Косинус числа. Работает быстрее стандартной функции COS	COSE(PI)	283
DEC D	Десятичный эквивалент шестнадцатеричного числа	DEC("9C40")	283
DPEEK P	Десятичное целое число, представленное в памяти по указанному адресу двумя байтами	DPEEK 40 000	268
EOF E	Работает только с Interface 1		–
FILLED F	Количество инвертированных пикселей последней операцией FILL	PRINT FILLED()	252
**HEX$** H$	Шестнадцатеричный эквивалент десятичного числа	HEX$(40000)	283

* Набор функций Beta Basic осуществляется с помощью ключевого слова FN и буквы, указанной в таблице ниже названия функции.

Функция, набор	Возвращаемое значение	Пример	Стр.
INARRAY U	Номер строки двумерного символьного массива, в которой обнаружено очередное вхождение строки-образца	INARRAY(A$(7)(3 TO 9),B$)	273
INSTRING I	Позиция в базовой строке, начиная с которой обнаружено очередное вхождение строки-образца	INSTRING (Start,A$,"TEST")	269
ITEM T	Тип очередного данного в списке DATA, который предстоит считывать оператору READ	ITEM()	263
MEM M	Количество свободной памяти для бейсик-программ	MEM()	243
MEMORY M$	Содержимое участка памяти, представленное символьной строкой	MEMORY$()(0 TO 23295)	268
MOD V	Остаток от деления двух чисел	MOD(A,B)	284
LENGTH L	Размер массива по заданному измерению	LENGTH(1,"A(1)")	276
NUMBER N	Целое число из диапазона 0...65535 — эквивалент двухбайтовой строки	NUMBER(CHR$ 55+CHR$ 28)	277
OR O	Результат поразрядного "ИЛИ" двух чисел	OR (A,B)	284
RNDM R	Случайное число в диапазоне от 0 до числа, указанного в аргументе	RNDM(175)	283
**SCRN$** K$	Символ (в том числе из набора UDG), стоящий в указанных строке и столбце экрана	SCRN$(10,5)	249
**SHIFT$** Z$	Строка символов, преобразованная по правилам в соответствии со значением первого аргумента	SHIFT$(3,"QWERTYqwerty")	270
SINE S	Синус числа. Работает быстрее стандартной функции SIN	SINE(PI)	283
**STRING$** S$	Приведенная строка символов, повторенная заданное количество раз	STRING$(N,"#")	269
**TIME$** T$	Текущее время	TIME$()	259

Функция, набор	Возвращаемое значение	Пример	Стр.
**USING$** U$	Символьное представление числа в соответствии с заданным форматом	USING$("##.###",D)	248
XOR X	Результат поразрядного "исключающего ИЛИ" двух чисел	XOR(L,M)	284

3. Перечень сообщений об ошибках Beta Basic

К стандартному перечню сообщений об ошибках (см. стр. 98) Beta Basic добавляет ряд собственных с кодами S...X. Меняются также ситуации, в которых появляются стандартные сообщения с кодами G и J:

G No room for line — нет места для строки

При перенумерации строк оператором RENUM сгенерированы номера уже существующих строк, не входящих в указанный интервал.

J Invalid I/O device — неверное устройство ввода-вывода

Обращение к окну с несуществующим номером при помощи операторов WINDOW или CLS.

S Missing LOOP — отсутствует LOOP

Не может быть найдено окончание цикла при использовании EXIT IF или при встрече DO, условие которого не предполагает ни одного прохода цикла.

T LOOP without DO — LOOP без DO

Применен оператор LOOP без DO либо осуществлен вход в тело цикла, минуя его заголовок.

U No such line — нет такой строки

В команде DELETE указан номер несуществующей строки.

V No POP data — нет данных для POP

Сделана попытка снять оператором POP данные со стека GO SUB, когда тот был пуст.

W Missing DEF PROC — отсутствует DEF PROC

Не найдено описание процедуры с указанным именем. Применен оператор LOCAL или END PROC без DEF PROC, либо осуществлен вход в тело процедуры, минуя его заголовок.

X No END PROC — нет END PROC

При попытке «перешагнуть» встреченное в тексте программы описание процедуры не может быть найдено ее окончание.

Значение, соответствующее коду ошибки, содержится в переменной ERROR, к которой имеется доступ из интерпретатора. Для сообщений с кодами 1...8 значение этой переменной равно самому коду, для сообщений с кодами от A до P — соответственно, 10...27. Для ошибок интерпретатора Beta Basic (коды S...X) значение ERROR равно 28...33. Для сообщений с кодами 0 и 9 переменная ERROR не определена.

4. Распределение памяти при работе с Beta Basic

P_RAMT (23732)	———	
	Символы, определяемые пользователем	
UDG (23675)	———	
	Интерпретатор Beta Basic	
	———	46960
	Область системных переменных для WINDOW, описание макрокоманд	
RAMTOP (23730)	———	
	...	
E_LINE (23641)	———	
	Переменные Бейсика	
VARS (23627)	———	
	Бейсик-программа	
PROG (23635)	———	

5. Использование Beta Basic с системой TR-DOS

Для того чтобы организовать корректную работу Beta Basic с TR-DOS, необходимо перед обращением к диску включать режим прерываний IM1, а затем возвращаться в IM2. Это можно делать, используя готовые процедуры в кодах, которые размещены в теле интерпретатора Beta Basic. Загрузка программы с диска, например, будет выглядеть так:

```
100 RANDOMIZE USR 63243: REM Включаем IM1
110 RANDOMIZE USR 15619: REM: LOAD "filename"
120 RANDOMIZE USR 61369: REM Включаем IM2
```

Впрочем, не обязательно каждый раз после обращения к диску возвращаться в режим IM2. Если в программе не используется оператор **CLOCK** и нет необходимости использовать клавишу **Break** для возврата в Beta Basic из процедур в кодах, то интерпретатор не замечает «подлога» и вполне корректно работает в IM1. Поэтому, как правило, достаточно в программе перед первым обращением к диску сменить режим прерываний на IM1, а в завершение ее работы восстановить режим IM2.

БЕЙСИК 128

Язык программирования Бейсик, «зашитый» в память ZX Spectrum 128, очень похож на стандартный Spectrum-Бейсик. В основном, отличие заключается в расширении функций редактора, а также в поддержке дополнительных аппаратных средств: музыкального процессора и электронного диска.

При включении ZX Spectrum 128 на экран выводится главное меню, предлагающее на выбор один из режимов работы компьютера: **Tape Loader**, **128 BASIC**, **Calculator**, **48 BASIC** и **Tape tester**. Войти в нужный режим можно, установив клавишами ↑ и ↓ голубую полоску на соответствующую позицию меню и нажав клавиши **Enter** или **Edit**. Коротко опишем эти режимы.

Tape Loader Единственное назначение режима — загружать программу с магнитофона, то есть выполнять самую популярную команду Бейсика LOAD "". Загружать можно как программы для Spectrum 128, так и для Spectrum 48. Но не все программы, написанные на стандартном Spectrum-Бейсике, после загрузки в этом режиме будут работать. При нажатии клавиши **Break** происходит возврат в главное меню.

128 BASIC После выбора этой позиции меню компьютер переходит в интерпретатор Бейсика 128*. Его мы подробно опишем далее.

Calculator Эта опция превращает компьютер в калькулятор, производящий математические расчеты с использованием всех операций и функций Бейсика. В режиме калькулятора работает экранный редактор, поэтому можно рассчитывать значения выражений уже находящихся на экране. В выражениях могут стоять не только числовые значения, но и переменные, ранее определенные в бейсик-программе. Задать переменную можно и в самом калькуляторе, выполнив, например, LET a=1. Расчет выражения производится после

* Так для краткости будем называть Бейсик, «зашитый» в ZX Spectrum 128.

нажатия клавиши **Enter**. Результат выводится в следующую за выражением строку. Для выхода из режима нужно нажать клавишу **Edit** и в появившемся меню из двух пунктов (**Calculator** и **Exit**) выбрать второй.

48 BASIC Работая в этом режиме, компьютер становится практически полностью совместимым с ZX Spectrum 48*. Выйти из режима **48 BASIC** можно, только выключив компьютер или нажав кнопку RESET.

Tape tester Не будем подробно описывать этот режим, отметим только то, что он предназначен для настройки магнитофона на запись/считывание информации. Возврат в главное меню осуществляется по клавише **Break**.

Если при включении ZX Spectrum 128 удерживать клавишу **Break**, то компьютер начинает выполнять тестовую программу, выйти из которой можно, только выключив его или нажав кнопку RESET.

КЛАВИАТУРА

Клавиатура ZX Spectrum 128 содержит 58 клавиш. Аппаратно она организована аналогично клавиатуре ZX Spectrum (и полностью совпадает с клавиатурой ZX Spectrum+). Дополнительные 18 клавиш лишь дублируют наиболее часто используемые сочетания пар клавиш. Например, нажатие клавиши **Edit** аналогично действию пары **CS/1**. Соответствие дополнительных клавиш стандартным парам приведено в табл. 22.

Таблица 22. **Дополнительные клавиши ZX Spectrum 128.**

ZX Spectrum 128	ZX Spectrum	ZX Spectrum 128	ZX Spectrum
True Video	CS/3	↓	CS/6
Inv Video	CS/4	←	CS/5
Delete	CS/0	→	CS/8
Graphics	CS/9	↑	CS/7
Edit	CS/1	.	SS/M
Extend Mode	CS/SS	,	SS/N
Caps Lock	CS/2	;	SS/O
Break	CS/Space	"	SS/P

* Некоторая несовместимость может возникнуть, например, из-за отличий в «прошивке» ПЗУ.

Следует отметить, что в режиме **128 BASIC** некоторые клавиши не выполняют приписанных им в стандартном Spectrum-Бейсике функций. Например, в этом режиме не работают клавиши **True Video** и **Inverse Video**, хотя они и имеются на клавиатуре.

В Бейсике 128 ключевые слова вводятся посимвольно (в том числе знаки <>, <= и >=). Символы **[**, **]**, **|**, \, {, } и ~ по-прежнему набираются через **Extend Mode** (**CS/SS**). Кроме того, в ключевых словах при вводе не делается различий между строчными и прописными буквами.

ЭКРАННЫЙ РЕДАКТОР

В редакторе Бейсика 128 нет явного деления экрана на основной (верхний) и служебный (нижний). Вводить и редактировать строки программы можно на всем пространстве экрана, точнее в верхних 22 его строках. Нижние две строки вспомогательные: в 23-й индицируется режим работы, в 24-й выводятся сообщения об ошибках. Впрочем, можно изменить режим и перевести редактор в две нижние строки экрана (см. дальше).

Знакоместо, в которое при нажатии клавиши будет помещен очередной символ, выделяется голубым курсором. Курсор перемещается по экрану клавишами ↑, ↓, ← и →. Удалить символ, стоящий перед курсором, можно, нажав клавишу **Delete**.

Поскольку ключевые слова Бейсика 128 вводятся по буквам, то нужно отделять их друг от друга пробелами. Цифры и знаки математических операций в программах также являются разделителями.

Набранная строка вводится клавишей **Enter**. Если у интерпретатора нет претензий к синтаксису строки, то раздается короткий звуковой сигнал и она выполняется или помещается в программу. Переход на другую строку с помощью команд редактора (о них дальше) также приравнивается к нажатию **Enter**.

Об обнаружении ошибки интерпретатор оповещает характерным звуком и помечает красным курсором позицию в строке, в которой ошибка найдена. И пока она не будет исправлена, интерпретатор не только не примет строку, но и не позволит выйти за ее пределы.

В Бейсике 128 нельзя ввести с клавиатуры коды непосредственного управления цветом (как в стандартном Бейсике в режиме курсора **[E]**).

К ZX Spectrum 128 предусмотрено подключение специальной дополнительной клавиатуры, делающей более удобной работу с редактором. Достать или сделать ее самому достаточно сложно, но не стоит расстраиваться: все возможности экранного редактора можно использовать, работая и только на основной клавиатуре. В табл. 23 приведены комбинации клавиш основной клавиатуры, при нажатии которых выполняются те или иные команды редактора.

Таблица 23. **Команды редактора Бейсика 128.**

Клавиша	Действие
Extend+P	курсор на 10 строк вверх
SS/I	курсор на 10 строк вниз
Extend+I	курсор на слово влево
Extend+SS/J	курсор на слово вправо
Extend+N	курсор в начало программы
Extend+T	курсор в конец программы
Extend+SS/2	курсор в начало строки
Extend+M	курсор в конец строки
Extend+SS/K	удалить символ под курсором
Extend+W	удалить слово справа от курсора
Extend+E	удалить слово слева от курсора
Extend+J	удалить символы от курсора до конца строки
Extend+K	удалить символы от начала строки до курсора
Extend+SS/8	аналогично функции Screen меню Бейсика 128

МЕНЮ БЕЙСИКА 128

Нажав клавишу **Edit**, можно из редактора войти в меню Бейсика 128. Кратко опишем пункты этого меню.

128 BASIC **Возврат к редактированию программы.**

Renumber **Перенумерация строк** программы, начиная с номера 10 и с шагом 10. Команда изменяет параметры в операторах перехода и вызова подпрограмм, но только при условии, что номера строк перехода заданы в явном виде (числом). Номер первой строки и шаг перенумерации можно менять, записывая с помощью оператора POKE другие значения в ячейки памяти с адресами, соответственно, 23444/45 и 23446/47. Приведем небольшую программу, которая задает параметры перенумерации:

```
10 POKE 23444,begin—256*INT(begin/256)
20 POKE 23445,INT(begin/256)
30 POKE 23446,step—256*INT(step/256)
40 POKE 23447,INT(step/256)
```

Переменная begin должна содержать номер первой строки, step — шаг перенумерации.

Если при заданных параметрах номер последней строки оказывается более 9999, то перенумерация не будет осуществлена.

Screen **Перевод окна редактора в две нижние строки** экрана. Опция предназначена для тех случаев, когда желательно сохранять в процессе редактирования информацию в верхних строках экрана (например, при графических построениях). Возвратиться к основному режиму работы редактора можно, повторно выбрав этот пункт меню.

Print **Распечатка на принтере листинга** бейсик-программы, находящейся в памяти.

Exit **Переход в главное меню ZX Spectrum 128** (текст бейсик-программы сохраняется).

ОПЕРАТОРЫ БЕЙСИКА 128

Список операторов Бейсика 128 дополнен относительно стандартного лишь двумя новыми операторами SPECTRUM и PLAY, кроме того, модифицированы некоторые старые.

Скорость выполнения программ в режиме **128 BASIC** несколько ниже, чем в режиме **48 BASIC**. Это связано с тем, что при выполнении многих операторов используются подпрограммы стандартного Бейсика и на переключение режимов уходит дополнительное, хотя и небольшое, время.

Ниже описаны новые операторы Бейсика.

SPECTRUM

Оператор SPECTRUM переключает компьютер в режим эмуляции ZX Spectrum 48, то есть выполняет действие, аналогичное опции **48 BASIC** главного меню. Отличие только в том, что при переходе с помощью оператора SPECTRUM программа, находящаяся в памяти, сохраняется. Если оператор SPECTRUM выполняется в программе, то она останавливается с сообщением 0 OK.

Кроме того, оператор SPECTRUM переопределяет канал, обслуживающий принтер. После выполнения оператора вся информация, поступавшая ранее на принтер, будет выводиться на экран.

PLAY

Оператор PLAY — основная достопримечательность Бейсика 128. Он обслуживает встроенный в ZX Spectrum 128 трехканальный музыкальный процессор*, на порядок увеличивающий звуковые возможности компьютера.

---

* Тип процессора — AY-3-8912.

Вслед за ключевым словом PLAY размещаются строки символов — *стринги*. Именно они задают музыкальную программу, то есть определяют, когда и что играть. В общем виде формат оператора записывается так:

PLAY a$[,b$,c$,d$,e$,f$,g$,h$]

– где a$, b$, ..., h$ — стринги. Как видно из формата, за ключевым словом PLAY может следовать от 1 до 8 стрингов. Однако сам музыкальный процессор пользуется только первыми тремя из них — a$, b$ и c$. Они задают программы, соответственно, для каналов A, B и C музыкального процессора. Эти программы выполняются одновременно, причем можно заставить звучать как все три канала, так и один или два, задав необходимое количество стрингов.

Остальные стринги (d$, e$, ..., h$) предназначены для управления музыкальными инструментами, которые можно подключить к ZX Spectrum 128 через специальный MIDI-интерфейс*.

Ноты ДО, РЕ, МИ, ФА, СОЛЬ, ЛЯ и СИ записываются латинскими буквами, соответственно, c, d, e, f, g, a и b. Например, оператор

PLAY "cdefgabC"

проиграет гамму ДО-мажор.

Ноты следующей октавы обозначаются прописными латинскими буквами C, D, E, F, G, A и B. Полутона задаются знаками: # — диез и $ — бемоль. Например, ДО-диез, запишется как #c, СИ-бемоль — $b, а гамма ДО-минор — cd$efg$a$bC.

Длительность звучания нот определяется числом от 1 (минимальная) до 9 (максимальная), стоящим перед нотой (табл. 24), например: PLAY "3G". Заданная длительность распространяется на все последующие ноты и паузы (см. команду задания паузы &).

Числами от 10 до 12 программируются триоли. Ноты триоли следуют непосредственно за числом: 3fed&11fed&fed. Триоли не изменяют установленной длительности нот.

Возможно сыграть ноту произвольной длительности. Для этого перед ней через символ подчеркивания (_) ставят несколько параметров, задающих требуемую длительность. Например, нота ДО длительностью 3/8 (1/8+1/4) запишется так: 3_5c. Длительность звучания последующих нот определяет последний параметр.

Кроме нот, в музыкальных программах могут использоваться специальные команды оператора PLAY:

- **Изменение октавы**. Номер октавы задается числом от 0 до 8 (по умолчанию установлена 5-я октава). Все ноты после этой команды будут звучать в выбранной октаве, пока она не будет сменена**.

  Диапазон звучания в переделах заданной октавы можно расширить, используя серии диезов или бемолей. Например, O6d можно записать как O4###Вю

* Musical Instrument Digital Interface – цифровой интерфейс для музыкальных инструментов.

** Заметим, что самые низкие ноты в 0-й и 1-й октавах без MIDI-интерфейса воспроизводиться будут неправильно.

Таблица 24. **Длительность звучания нот в операторе PLAY.**

Число	Нота	
Одиночные ноты		
1	Одна шестнадцатая	♪
2	Одна шестнадцатая с точкой	♪.
3	Одна восьмая	♪
4	Одна восьмая с точкой	♪.
5	Одна четвертая	♩
6	Одна четвертая с точкой	♩.
7	Одна вторая	𝅗𝅥
8	Одна вторая с точкой	𝅗𝅥.
9	Целая	𝅝
Триоли		
10	Одна шестнадцатая	♬
11	Одна восьмая	♬
12	Одна четвертая	♩♩♩

N **Разделитель параметров**. Он используется в том случае, если нужно задать длительность звучания нот сразу за командой с числовым параметром: O6N4gD; хотя тоже самое допустимо записывать и с пробелом: O6 4gD.

& **Пауза установленной длительности**: 6c&C (здесь длительность паузы равна 6).

V **Громкость звучания**. Громкость задается параметром, следующим за командой: от 0 (минимальная — выключено) до 15 (максимальная). Минимальная громкость (команда V0) может использоваться для временного отключения канала.

W **Программирование эффектов**. Ноты могут воспроизводиться не только с фиксированной громкостью, но и со всевозможными эффектами: затуханиями, всплесками и т. д. Характер эффекта задается числом от 0 до 7, в соответствии с табл. 25.

X **Временной параметр звукого эффекта**. Для эффектов 0...3 параметр задает длительность действия эффекта, для 4 и 5 — период, для 6 и 7 — полупериод. Значение параметра выбирается из диапазона 0...65535. Для эффектов 0...3 оптимальное значение — около 1000, для 4...7 — около 300. По умолчанию значение параметра принято равным 65535.

U **Включение звукового эффекта**. После этой команды все ноты будут воспроизводиться с эффектом, установленным командой W. Эффекты отключаются по завершении стринга либо при изменении громкости (команда V). Следующая программа продемонстрирует действие всех эффектов:

```
10 FOR i=0 TO 7
20 PLAY "UX1000W"+STR$ i+"cdef&"
30 NEXT i
```

Таблица 25. **Программирование звуковых эффектов.**

Значение параметра	Эффект	Диаграмма
0	единичный спад, затем тихо	
1	единичный подъем, затем тихо	
2	единичный спад, затем громко	
3	единичный подъем, затем громко	
4	повторяющийся спад	
5	повторяющийся подъем	
6	повторяющийся подъем-спад	
7	повторяющийся спад-подъем	

Параметр команды X

T **Темп исполнения**. Задается числом в интервале от 60 до 240. По умолчанию устанавливается темп, соответствующий команде T120. Задать темп можно лишь для всей мелодии в целом, поэтому он определяется только в стринге канала A (в других каналах команда T игнорируется).

() **Повтор музыкальной фразы**. Музыкальная фраза, взятая в скобки, повторится еще раз: (ec)fe7d. Допустимо использовать не более 4 пар скобок.

Закрывающая скобка без соответствующей ей открывающей заставит музыкальную фразу повторяться бесконечно от начала стринга. Используется это, например, в басовых партиях.

H **Останов оператора PLAY**. Встретив эту команду в стринге любого канала, оператор PLAY закончит свою работу. Команда

используется, например, для выхода из «зацикленной» басовой партии, когда заканчивается основная мелодия.

! **Ограничитель комментариев.** В стринг можно вставлять комментарии — любой текст, ограничивая его знаками !. Например:

```
PLAY "#c!cis!$d!des!"
```

Знак ! в конце музыкальной программы ставить не обязательно.

M **Режим работы каналов.** Каждый канал может не только воспроизводить чистый тон, но и одновременно генерировать так называемый *белый шум*. Распределение режимов задается командой M с параметром в интервале 1...63. Значение параметра определяется суммой кодов, соответствующих режиму каждого из каналов (табл. 26).

Таблица 26. **Режимы работы каналов.**

Канал	A	B	C
Тон	1	2	4
Шум	8	16	32

В следующем примере канал A настраивается на тон, а канал B — на шум:

```
PLAY "M17cegbdfaCH","O3cC)"
```

Y **Установка канала MIDI-интерфейса.** Число от 1 до 16, следующее за командой, задает номер канала интерфейса, в который необходимо направить вывод музыкальных данных.

Z **Передача управляющих кодов MIDI-интерфейсу***.

ЭЛЕКТРОННЫЙ ДИСК

В дополнительных 80 килобайтах памяти ZX Spectrum 128 организован электронный диск, позволяющий оперировать несколькими программами без обращения к внешним носителям. Естественно, программы должны быть предварительно на этот диск записаны в виде файлов. Особенностью файловой структуры электронного диска является то, что на него не могут быть записаны файлы разных типов с одинаковыми именами. Емкость электронного диска более 400 файлов. Перед выключением или сбросом компьютера файлы с электронного диска нужно переписать на магнитный носитель. Действие команды Бейсика NEW не затрагивает электронный диск.

---

* Подробнее см. в описании MIDI-интерфейса [8].

Операторы, поддерживающие работу электронного диска:

SAVE!

Запись файла на электронный диск.

LOAD!

Считывание файла с электронного диска.

MERGE!

Подгрузка к программе файла с электронного диска.

Эти три оператора похожи на инструкции работы с магнитофоном SAVE, LOAD* и MERGE и имеют аналогичный синтаксис.

CAT!

Вывод на экран каталога электронного диска (списка всех файлов). Файлы выводятся в алфавитном порядке. К сожалению, в каталоге не указываются типы файлов.

ERASE!

Удаление файла с электронного диска. Имя файла записывается вслед за ключевым словом без указания типа файла. Например, оператор ERASE! "Name" удалит файл с именем "Name" (независимо от его типа). Если файлов много, то выполнение этой команды может занять несколько минут!

РАБОТА С ПРИНТЕРОМ

Еще одно отличие Бейсика 128 от стандартного — поддержка другого типа принтера. Операторы печати, такие как LPRINT, LLIST и COPY, работают с Epson-совместимыми моделями принтеров. Подключается принтер с помощью встроенного в Spectrum 128 последовательного интерфейса RS-232.

Скорость передачи данных по последовательному интерфейсу устанавливается оператором:

FORMAT "p";<speed>

Параметр <speed> выбирается из ряда: 50, 110, 300, 600, 1200, 2400, 4800, 9600 и должен соответствовать скорости обмена в бодах (бит/сек.), установленной в принтере. При запуске компьютера устанавливается скорость 9600 бод.

---

* Символьный массив перед загрузкой с электронного диска должен быть определен оператором DIM. Иначе он будет загружен как простая строка символов. Это замечание справедливо и для загрузки с магнитофона символьного массива в режиме **128 BASIC**.

ПРИЛОЖЕНИЯ

1. Сообщения об ошибках интерпретатора Бейсик 128

Интерпретатор Бейсика 128 в дополнение к сообщениям об ошибках стандартного Бейсика имеет и свои собственные:

a MERGE error

не может быть выполнена команда MERGE!.

b Wrong file type

несоответствие найденного и ожидаемого типа файла в команде, работающей с электронным диском.

c CODE error

заданная в команде VERIFY!"NAME"CODE <start>,<length> длина (<length>) больше реальной длины файла.

d Too many brackets

превышено число скобок в операторе PLAY.

e File already exists

файл с указанным именем уже существует на электронном диске.

f Invalid name

неправильное имя файла (либо пустое, либо более 10 символов).

g File does not exist

файл с заданным именем не существует на электронном диске.

h File does not exist

файл с заданным именем не существует на электронном диске*.

i Invalid device

в команде FORMAT задано неправильное имя устройства.

j Invalid baud rate

в команде FORMAT задано нулевое значение скорости передачи по последовательному интерфейсу.

k Invalid note name

в одном из стрингов оператора PLAY встретилась неопознанная команда.

---

* Это сообщение, очевидно, аналогично сообщению с кодом **g**, однако ссылки на него в ПЗУ найти не удалось.

l Number too big

в одном из стрингов оператора PLAY указано недопустимое значение параметра.

m Note out of range

нота не может быть воспроизведена музыкальным процессором (например, нота C в 8-й октаве).

n Out of range

числовой параметр в команде музыкальной программы слишком мал или велик.

o Too many tied note

слишком много параметров, задающих длительность, связано знаком _.

2. Системные переменные ZX Spectrum 128

В ZX Spectrum 128 расширена область системных переменных. Сделано это за счет использования части оперативной памяти, отведенной в ZX Spectrum 48 под буфер принтера. Ниже приводятся название, адрес и краткое описание некоторых системных переменных, значения которых могут быть использованы из Бейсика.

23389 (BANKM)

копия внутреннего системного регистра, определяющего текущую архитектуру компьютера (из Бейсика изменять не рекомендуется).

23391 (BAUD)

скорость обмена по последовательному интерфейсу. Хранится в «обратном» виде, то есть сначала записан старший байт, затем младший.

23395 (COL)

позиция (номер столбца) вывода на принтер.

23396 (WIDTH)

количество печатных позиций на принтере. При достижении этого числа программа печати автоматически переводит строку.

23429/30/31 (SFSPACE)

количество свободных байтов на электронном диске.

23444/45 (RNFIRST)

номер строки начала при перенумерации строк. Хранится в «обратном» виде.

23446/47 (RNSTEP)

шаг перенумерации. Хранится в «обратном» виде.

Список литературы

1. Ларченко А. А., Родионов Н. Ю. ZX Spectrum для пользователей и программистов — СПб.: Импакс, 1991.
2. Бленд Г. Основы программирования на языке Бейсик в стандарте MSX: Пер. с англ. — М.: Финансы и статистика, 1989.
3. Геворкян Г. Х., Семенов В. Н. Бейсик — это просто. — М.: Радио и связь, 1989.
4. Кергаль И. Методы программирования на Бейсике (с упражнениями): Пер. с фр. — М.: Мир, 1991.
5. Очков В. Ф., Пухначев Ю. В. 24 этюда на Бейсике — М.: Финансы и статистика, 1988.
6. Уолш Б. Программирование на Бейсике: Пер. с англ. — М.: Радио и связь, 1987.
7. Фокс А., Фокс Д. Бейсик для всех: Пер. с англ. — М.: Энергоатомиздат, 1986.
8. Schulze, Hans-Jochen/Engel, Georg: Moderne Musikelektronik, Praxisorientierte Elektroakustik und Gerate zur elektronischen Klangerzeugung. — Berlin: Militarverlag der DDR (VEB), 1989.

ОГЛАВЛЕНИЕ

Для оформления заказа на книги и программы

необходимо перевести по почте (для частных лиц) или через банк (для организаций) на расчетный счет фирмы «Питер» соответствующую сумму. Копия платежного документа с указанием перечня заказываемых товаров отправляется в адрес фирмы. Не забудьте сообщить Ваш обратный адрес.

Цены приведены без стоимости доставки. Почтовые расходы оплачиваются заказчиком при получении бандероли. В случае изменения цен доплата производится также при получении бандероли.

Фирма «ПИТЕР»

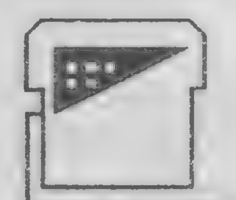
Расчетный счет для иногородних платежей:

к/с 41000161499 р/с 49100609294
в банке ЦРКУ ГУ ЦБ России
МФО 161002

Расчетный счет для платежей из С.-Петербурга:

р/с 49100609294 в Парнасском отд.
АО «Банк Санкт-Петербург»
С.-Петербурга, МФО 171337

Адрес: 196244, С.-Петербург, а/я 21



Телефон/факс: **(812)-235-3749**

E-mail 2:5030/25 @ fidonet.org

По вопросам приобретения литературы можно также обращаться к нашим региональным диллерам:

г. Томск, 634045, ул. 19-й Гвардейской дивизии, 13,
Фирма «Старт Ltd.», тел. 44-44-93, 44-80-78

г. Смоленск, 214018, Киевский пер. 16,
Коммерческий центр «Апекс», тел. 6-25-19

Украина, г. Херсон, 325000, ул. Суворова, 3,
Учебно-научное объединение «Дисплей», тел. 4-01-52

Украина, г. Луцк, 263026, а/я 94, Фирма «Скиф», тел. 5-99-12

Беларусь, г. Минск, 220005, а/я 95,
НПК «Протон-Запад», тел. 63-52-72, 33-13-11



**ОТКРЫТАЯ
МНОГОКАТАЛОГОВАЯ
ДИСКОВАЯ ОПЕРАЦИОННАЯ СИСТЕМА**

isDOS 1.0

Операционная система isDOS устанавливается на все типы компьютеров, совместимых с ZX Spectrum и согласуется с внешними устройствами такими, как электронный диск, дисководы 5.25 и 3.5 дюйма, винчестер, модем и локальная сеть.

Оконный интерфейс, контекстно-зависимый встроенный HELP, работа с командными файлами, системная дата, маска, собственное меню в каждом каталоге и многое другое — понравятся самому требовательному пользователю.

Многооконный текстовый редактор, встроенный в ОС isDOS, работает с файлами в формате Multi-Edit и Lexicon, в нескольких режимах отображения с длиной строки в 252 символа. Размер обрабатываемого текста ограничен только емкостью дискет. Печать текстов возможна различными шрифтами, с разбивкой на страницы и колонки, с автоматической нумерацией. Возможна раздельная печать четных и нечетных страниц, печать графических полутоновых изображений в произвольном месте листа.

ОС isDOS предоставляет возможность создания динамических рекламных текстовых заставок.

Под isDOS созданы различные АРМы со структурой данных в формате DBASE.

ОС isDOS значительно расширяет возможности использования компьютеров типа ZX Spectrum в научной деятельности, в сфере бизнеса и управления, для создания прикладных и коммерческих программных продуктов.

ОС isDOS ориентирована на пользователя, не обладающего навыками программирования, однако не оставит равнодушным и профессионального программиста.

Фирма «IskraSOFT» при разработке isDOS ориентировалась на стиль и идеологию операционной системы MS-DOS, устанавливаемой на компьютерах типа IBM PC. Такой подход имеет неоспоримое преимущество — пользователь, работающий с isDOS на ZX Spectrum при переходе на IBM PC попадает в знакомую, привычную ему среду.

ОС isDOS — новая жизнь ZX Spectrum!

IskraSOFT Publicity Division 1992
194100, Санкт-Петербург, а/я 34
(812)-245-18-97

КАКОЕ ВПЕЧАТЛЕНИЕ ОСТАВИЛА У ВАС КНИГА

«Диалекты Бейсика для ZX Spectrum»

Точность изложенных сведений	(1...5)	
Новизна информации	(1...5)	
Стиль изложения	(1...5)	
Удобство в пользовании	(1...5)	
Оформление	(1...5)	
Общая оценка	**(1...5)**	

Как бы Вы оценили каждую главу в отдельности

Spectrum-Бейсик	(1...5)	
Компиляторы Spectrum-Бейсика	(1...5)	
PRO-DOS	(1...5)	
Laser Basic	(1...5)	
MegaBasic	(1...5)	
Beta Basic	(1...5)	
Бейсик 128	(1...5)	

(отрежьте и приклейте на конверт)

Куда 196244, Санкт-Петербург,

а/я 21

Кому Фирма «Питер»

Какими диалектами Бейсика Вы пользуетесь чаще всего

Описание каких программ и аппаратных средств для ZX Spectrum Вы посоветуете включить в наши следующие издания

Хотите ли Вы стать нашим региональным распространителем литературы по ZX Spectrum

Ваш возраст, профессия

Сколько лет работаете с ZX Spectrum

Ваше имя, адрес, телефон

Если у Вас есть предложения и пожелания, можете изложить их

Большое спасибо всем, кто прислал свои отзывы на книгу «ZX Spectrum для пользователей и программистов»

